Knowledge Graph-Driven Conversational Agents

Joseph Bockhorst, Devin Conathan, Glenn Fung
Machine Learning Research and Innovation
American Family Insurance
Madison, WI
{jbockhor,dconatha,gfung}@amfam.com

Abstract

We present an approach to developing task-oriented conversational interfaces that addresses two key challenges that designers face: i) constructing a system grammar that strikes the right balance between the expressiveness necessary to carry out the task and the ability to correctly infer parses from natural language and ii) dealing with parse ambiguities. We address the latter of these with an approach to semantic parsing where the system constructs a semantic parse progressively, throughout the course of a multi-turn conversation in which the system's prompts to the user derive from parse uncertainty. We construct the system grammar by leveraging the structured types and entities of an underlying knowledge graph (KG) complemented by a machine learning (ML) driven restructuring procedure. The rationale for inducing the grammar from KG types stems from our perspective that the aim of a conversation in our system is to identify a single instance of an *a priori* unknown type in the KG. ML models provide suggestions for adding additional structure to the KG (and thus a re-structuring of the grammar). We provide results of an empirical study into the value of our human-in-the-loop knowledge graph restructuring algorithm showing that meaningful structure can be recovered by applying machine learning to existing data from two case studies in the insurance domain.

1 Introduction

Modern conversational agents employ elements from both the symbolic and non-symbolic sides of artificial intelligence. On the symbolic side consider that typically the aim of an agent is, given natural language inputs from a human user, to infer a representation understood by the system and faithful to the user's intentions. Furthermore, the understandings available to the system correspond to semantic parses in the system's grammar. In contrast to these symbol heavy components, the most effective means of producing these semantic parses are dominated by non-symbolic models produced by machine learning techniques.

Here we address key challenges in conversational interfaces concerning constructing the system grammar and dealing with parse ambiguity by formalizing the symbolic language and behavior of agents and leveraging machine learning techniques to design and curate the underlying systems.

We define a **knowledge graph-driven conversational agent (KGCA)** as a type of interface whose behavior is determined in part by a knowledge graph. See Figure 1 for an overview. The job of the agent is to interact with the user to translate user inputs (over multiple turns if necessary) into a semantic parse in the system's grammar that is in agreement with the user's intent (right side of Figure 1). Machine learning plays a crucial role in the translation step when user inputs are natural language. Since the effectiveness of the agent depends critically on the knowledge graph, we propose a human-in-the-loop method for curating the KG structure to best achieve the KGCA's goals. A

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

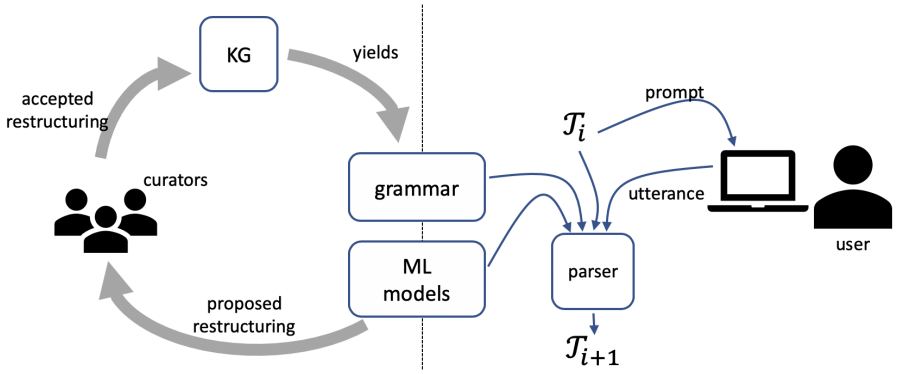

Figure 1: KGCA overview. The left side diagram represents our human-in-the-loop restructuring approach. ML trained models are used to propose restructurings of the knowledge graph to curators. Accepted restructurings cause subsequent changes to the system grammar. The right side shows changes to the semantic parse for one turn of the conversation. Inspection of the incomplete semantic parse tree $\mathcal{T}_i$ after i turns is used to produce a prompt to the user who responds with an utterance. The tree and utterance together with the grammar and ML models extend the tree to produce $\mathcal{T}_{i+1}$ (see the appendix for the pseudo-code that shows how to update $\mathcal{T}_i$).

machine learning algorithm iteratively proposes candidate alterations of the knowledge graph which can be accepted or rejected by a curator (left side of Figure 1).

We detail case studies of applying our methods for two quite distinct tasks in the auto and property insurance industry. The first is a customer facing conversational agent that allows customers to initiate the claims process by submitting a "first notice of loss" (FNOL) that contains the information necessary to route the claim, for example, to a claims adjuster with the training needed for that particular claim. The second case study is an internal facing virtual assistant that retrieves responses to frequently asked questions (FAQs) about business products and processes.

Our contributions are threefold: i) a method for inducing conversational agent grammars from the type structure of a knowledge graph, ii) an approach for progressively building up a semantic parse for complex concepts over the course of a multi-turn conversation, and iii) a novel, simple and effective human-in-the-loop approach to knowledge graph restructuring that uses inspection of ML model-derived embeddings of training utterances to propose restructuring operations to the curator.

2 Related Work

Semantic parsing is the the task of mapping natural language to a domain-specific meaning representation, such as a logical form or a parse tree in a context-free grammar. We briefly compare the work presented here with related efforts.

Typically investigations into semantic parsing focus on the "single-shot" setting in which a full meaning is inferred from a single input or sentence [1, 9, 4, 3]. Related approaches construct semantic parses in the context of a conversation [5, 8]. While these context-aware approaches use the conversation history to help infer meaning, for example by helping resolve linguistic references like pronouns, they are still single-shot. There has been much less focus on interactive settings like ours that construct meaning representations over multiple conversational turns, but there has been some recent work [10]

3 Knowledge Graphs and Conversational Agents

3.1 Entities, Properties and Types

A knowledge graph is a way of storing information that is particularly useful for conversational agents. Let $\mathcal{E}$ be a set of **entities** and $\mathcal{P}$ be a set of **properties**. A **knowledge graph** $\mathcal{K} \subset \mathcal{E} \times \mathcal{P} \times \mathcal{E}$ represents a set of facts; if $(e_1, p, e_2) \in \mathcal{K}$, then entity e_1 has property p equal to e_2 (e.g. (`Barack Obama`, `born in`, `Honolulu`)).

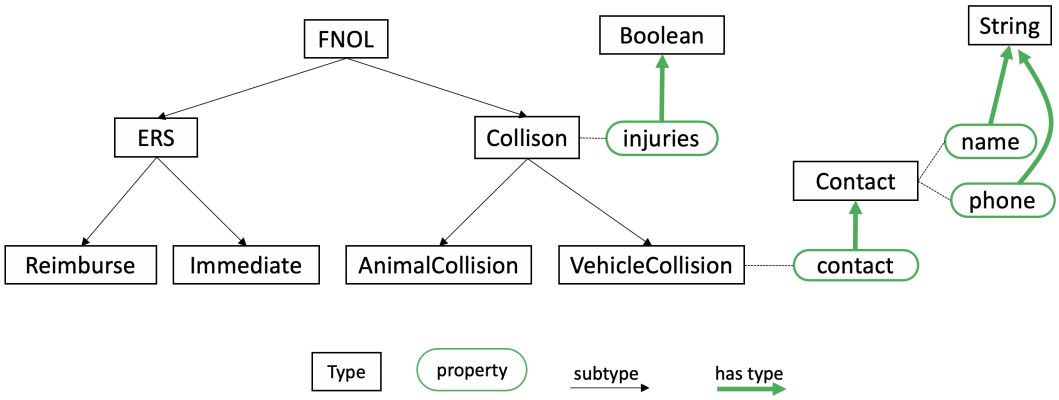

Figure 2: A simplification of a fragment of the knowledge graph for the FNOL case.

$\mathcal{E}$ contains both entity instances and entity types. An entity's type is expressed in the graph with a special property `type`: (`10`, `type`, `Number`), which says there's an entity `10` that's an instance of the type `Number`. We can also have a special property `subtype` for capturing type hierarchies: (`Number`, `subtype`, `Integer`).

$\mathcal{E}$ usually contains basic types like `String`, `Number` and `Boolean` and custom types specific to a domain (e.g. `Vehicle`). $\mathcal{K}$ includes schema information about these custom types; for instance, (`Vehicle`, `year`, `Integer`) indicates that entities with type `Vehicle` have a property `year` that is a `Integer`. We call the type hierarchy and associated schemas the **knowledge graph structure**. The knowledge graph structure defines what kinds of entity instances exist in the graph and (through type hierarchies and properties) the relationships that might exist between them.

We define a **conversational episode** as a process that involves identifying an instance of a given type. An KGCA is associated with an abstract type (an abstract type cannot have any instances) called the root type for the agent. The KGCA aims to identify an instance of a concrete subtype of its root. Presently we support two kinds of episodes: a **retrieval** episode aims to identify an instance that exists in the KG while a **construction** episode creates a new instance by specifying values for all properties according to its schema.

3.2 Constructing the Episode Grammar

We now describe how we use the KG to construct the KGCA's meaning representation in the form of a context-free grammar we call the **episode grammar** (EG). Starting from the root type we form the EG so that its sentences correspond to a single instance of a concrete subtype of the root.

We refer to Figure 2 to us help illustrate our approach. For simplicity of exposition we assume that all non-final types T (that is, a type that has subtypes) are abstract. In other words, a type T is concrete if and only if it is a final type. Relaxations to the more general case require more bookkeeping but otherwise add little additional complexity.

The nonterminals of the EG are composed of types and properties from the EG. Each type in the type hierarchy under the root contributes a nonterminal to the EG. Given non-final type T we create a production rule for each subtype of T. For example for the types `FNOL` and `Collision` we have

```
      FNOL  →  ERS | Collision
 Collision  →  AnimalCollision | VehicleCollision
```

The mapping for final type T with properties $p_1 \ldots p_m$ is a single production rule $T \rightarrow T.p_1 \ \ldots \ T.p_m$. The final type `VehicleCollision`, for example, yields

```
VehicleCollision  →  VehicleCollision.injuries  VehicleCollision.contact
```

Note that a type's properties include the properties of its supertypes. Each property $T.p_i$ has a single production $T.p_i \rightarrow \textsc{Type}(T.p_i)$ mapping to its type $\textsc{Type}(T.p_i)$. If $\textsc{Type}(T.p_i)$ is not in the EG it is

also added using the method for adding types we just described. Continuing our example, expanding out the properties of `VehicleCollision` yields

```
          VehicleCollision  →  VehicleCollision.injuries  VehicleCollision.contact
 VehicleCollision.injuries  →  Boolean
  VehicleCollision.contact  →  Contact
                   Contact  →  Contact.name Contact.phone
              Contact.name  →  String
```

For more details on the KGCA and the role of the episode grammar we refer you to the appendix.

3.3 Case study: FNOL

Here we describe a KGCA designed to assist customers at the beginning of the claims process. The motivating business need is an easy-to-use, automated process for customers to initiate a claim that captures information need to internally route new claims (such as to the appropriate class of claims adjuster) as well as collect certain key pieces of information (such as contact information for third parties). Using existing enterprise artifacts relating to the "loss cause" taxonomy a KG was constructed containing 21 subtypes of `FNOL`, the root type for this application, and 50 total types (which includes the types of properties as well). Figure 2 shows a fragment of the KG for this case. Important types and properties were associated with entity recognizes as well as canonical names and other annotations to assist user interface design.

The business requested that the application begin with an open ended prompt such as "Please describe what happened" for customers to enter an unstructured text description of their incident. Following that the aim is to efficiently gather the remaining needed information. Given the level of detail it is unlikely that all required information is contained in the initial text (e.g. sometimes we need to know if the airbags were deployed); this motivates our multi-turn approach to semantic parsing.

These requirements called for mixed conversation strategy that uses a combination of variable and fixed increment (see Section A.3 in the appendix). At the start of an episode a variable strategy is used for the initial prompt and then a fixed strategy takes over. With this hybrid approach, after a user to enters an initial phrase such as “I got rear ended” the system is able to infer that a node for the final type `VehicleCollision` must be in the parse tree, filling in other components of the tree that can be inferred, and enabling more precise follow up questions.

A key advantage of this general approach of driving a conversation agent from a knowledge graph is that it does not require business users or IT professionals to author explicit workflows since the workflow is dynamically generated from the graph.

4 Knowledge graph restructuring

A KGCA can fail in a number of ways, but one of the main causes is that the system is unable to map the user’s inputs to the appropriate parse due to limitations of natural language understanding. Here we introduce the idea of restructuring the knowledge graph (i.e. changing the type hierachy and schemas) in order to improve the performance of a KGCA. With more structure the task becomes simpler because it can be achieved by successively narrowing down the set of possible instances (with fixed or or variable increment strategies) by parsing more information about the user’s desired type and properties.

We propose leveraging machine learning techniques to impose, discover or suggest optimal structures. This primarily applies to retrieval episodes where you have a given set of instances. We leave extending knowledge graph restructuring to construction retrieval for future work.

4.1 Changing the type hierarchy

We formalize a change to the type hierarchy as one of three operations:

- A **merge** of two types, which creates a common supertype
- A **reassign** of a type, which assigns it a new supertype

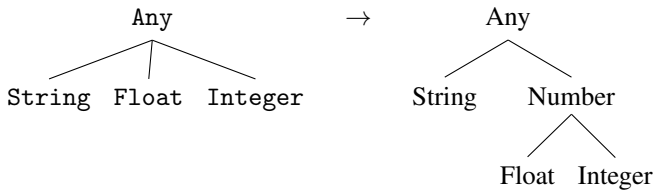

Figure 3: An example merge operation.

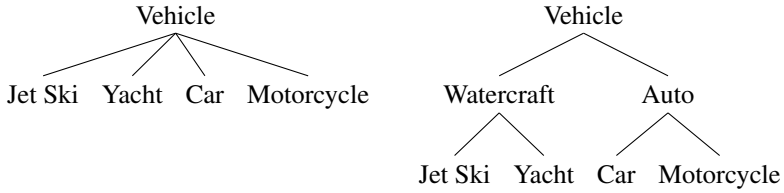

Figure 4: Examples of flat and structured vehicle hierarchies.

- A **split** of a type into one or more subtypes (this subsequently partitions the original type's subtypes and instances)

Examples: One merge operation might assign the types `Float` and `Integer` a common supertype `Number` (see Figure 3). A reassign might designate that the type `URL` is actually a subtype of `String`. Finally, a split might partition the type `Vehicle` into `Car` and `Motorcycle`, and each vehicle would now be an instance of exactly one of `Car` or `Motorcycle` (assuming cars and motorcycles are the only instances of vehicles in our graph).

As part of a merge operation, you can add properties to the types in order to distinguish them. In the `Vehicle` case, one could create the `number_of_wheels` property for all `Auto` instances. All `Car` instances have 4 wheels and all `Motorcycle` instances have 2 wheels. We now have a property we can inquire about in order to narrow down the set of possible `Vehicle` instances for a retrieval episode: "How many wheels does the vehicle have?". This process of refinement by describing properties is usually only necessary when many candidate instances remain; often one can simply enumerate the possibilities: "Is it a car or motorcycle?".

For this paper we will assume we are starting with a flat hierarchy of types and wish to build a more complex structure, so we will be focusing on the merge and reassign operations. We leave exploring approaches to using machine learning for optimal split operations for future work.

4.2 Structure and KGCA performance

The structure of the KG directly impacts the performance of the KGCA. As a concrete example, assume we are trying to determine an instance of vehicle from the utterance *"I got hit on my way to work today"*, and consider the consequences of using the two vehicle type hierarchies specified in Figure 4. An NLP recognizer will struggle to determine if this user is referring to a `Car` or `Motorcycle` because the utterance is ambiguous; however, it is unlikely that the user is referring to a `Watercraft`. Accordingly, the KGCA can parse the utterance as an `Auto` type and ask a followup refining question to produce an instance or further refine the type.

4.3 Human-in-the-loop restructuring

Developing the KG structure can be a daunting task. It is sometimes extremely difficult to start the restructuring process; with a flat hierarchy of n types, there are $2 \times n \times (n-1)$ possible merges and reassigns to consider (as of writing, our virtual assistant detailed in Section 5 has 313 possible responses and 97656 possible merges!).

We propose using machine learning to suggest possible operations in order to prioritize likely candidates. This approach assumes you have labeled utterances $U_c \subset U$ (where U is the set of

all utterances) for the each final type (or class) $c \in C$ of the tree and can train a classifier. After training the classifier, we produce an encoder function $f : U \to \mathbb{R}^d$. Any classifier that can be used to represent utterances as d-dimensional vectors (for some d) can be used here.

Next we use the encoder to produce representations for each class. If U_c is the set of all utterances associated with class c, then

$$\mu_c = \frac{1}{|U_c|} \sum_{u \in U_c} f(u)$$

is the class centroid, or mean vector representation of that class. We then suggest the two classes with the closest centroids as candidates to merge or reassign:

$$c_1^*, c_2^* = \underset{\substack{c_1, c2_2 \in C \\ c_1 \neq c_2}}{\arg\min} D(\mu_{c_1}, \mu_{c_2})$$

where D is some distance function (we use Euclidean distance). The user determines if these two classes should be merged or if one should assigned as the subtype of the other. If they are, the appropriate changes to the type hierarchy are made and utterances are reassigned to their new classes. If applicable, the user can create new properties and refinement prompts to help distinguish between different subtypes of a type. In a real application (see Section 5), it is often impractical and unnecessary to retrain the classifier for each iteration.

Algorithm 1 Human-in-the-loop restructuring pseudocode

$L = \{\}$ ▷ initialize labeled pairs
repeat
  $f \leftarrow$ TRAINENCODER()
  for all $c \in C$ **do**
    $\mu_c \leftarrow \frac{1}{|U_c|} \sum_{u \in U_c} f(u)$
  end for
  $c_1^*, c_2^* \leftarrow \arg\min_{\substack{c_1, c2_2 \in C \\ c_1 \neq c_2 \\ (c_1, c_2) \notin L}} d(\mu_{c_1}, \mu_{c_2})$
  ORACLE(c_1^*, c_2^*) ▷ ask curator if these should be merged or reassigned
  $L \leftarrow L \cup \{(c_1^*, c_2^*)\}$
  $C,\ U \leftarrow$ UPDATE(C, U, L) . ▷ update classes and training utterances to reflect changes
until Users stops

5 Case study: Frequently Asked Questions

5.1 Background

At our company, we have a virtual assistant chatbot whose purpose is to answer common questions about products and processes. Questions can range from broad to specific and can include domain-specific jargon. Some typical questions are:

- "How do I list the youthful driver as a non-driver?"
- "Do condos have to meet the minimum deductible?"
- "How do I change the term on this policy?"

As of writing, the virtual assistant has 313 possible responses which range from single sentences to many paragraphs or links to external documentation. We cast this chatbot as a KGCA by creating 313 instances of the `FAQ` type, which is a retrieval conversational episode; that is, the goal of the chatbot is to map the user's query utterance to the appropriate `FAQ` instance.

We have 8315 total training utterances. We use a CNN with max pooling and pretrained Glove embeddings trained on the Common Crawl 840B dataset [6] [7]. By applying our CNN classifier as a straightforward 1-of-k document classification task, we are able to achieve 78% accuracy on a holdout test set (10:1 train/test split). This performance is not satisfactory for the needs of our users,

so we seek to increase accuracy by creating `FAQ` subtypes and organizing the instances as subtypes and introduce properties which can be used to refine and disambiguate between instances.

Fortunately, there is a lot of inherent structure to the `FAQ` instances. Most can be grouped under general categories like "claims" or "discounts". Some responses are identical except they differ in one small aspect. For example, users can ask questions about two lines of product; the response to the question "Is there a special limit for cameras?" depends on the product line, but users rarely specify which line they are asking about in their utterances.

Our goal is to bring structure to the questions by placing them in categories which correspond to supertypes in Section 3.2. This way, if a complete parse is not possible due to ambiguity of the utterance (e.g. they did not specify the product line), the KGCA can achieve a partial parse by determining the response the user is seeking is in the *camera* category. The KGCA can then complete the parse by asking the user to fill in the `product_line` property.

5.2 Procedure

Without any special tools or algorithms, subject matter experts have grouped the 313 responses into 161 categories. We consider the mapping of the 313 responses to the 161 categories our "gold structure" for the sake of this case study, and we examine how much of this structure can be recovered using our approach detailed in Section 4.3. We use the gold structure as an oracle to simulate the user in the human-in-the-loop process. We use a slightly modified version of the process described in 4.3. Roughly the process is:

1. Train the encoder
2. Suggest the closest class pair (if pair is not a previously seen true or false positive)
3. If k responses have been received since last time encoder was trained, go to step 1.
4. If user accepts:
   Pair labeled a true positive
   Merge or reassign
   Go to step 2.
5. If user rejects:
   Pair labeled a false positive
   Go to step 2.

Note that the gold structure only contains a 2-level hierarchy (root $\rightarrow$ category $\rightarrow$ response). This simplifies the procedure and simulations because we can infer whether an operation should be a merge (if the two candidates are responses) or a reassign (if one of the candidates is a category).

5.3 Analysis

It takes 152 operations to get from a flat hierarchy to the gold structure, so this is the maximum number of true positives possible. We report the number of true positives and false positives for each iteration with $k = 10$. For the first few iterations the procedure achieves perfect precision and all 10 suggested pairs were merged into new classes. Unsurprisingly, as the number of iterations increases the task becomes more difficult as there are fewer positives to find. We find that with a reasonable amount of time from the user (250 responses), 70% of the original structure can be recovered. The results of the experiment are summarized in Figure 5.

Interestingly, we found that a majority of the false positives were actually reasonable suggestions overlooked by our subject matter experts. For example, there is a whole category of responses called "small talk" for when users inquire about the age of the chatbot or where it lives; these were not placed into the same category but were often suggested by our algorithm. We also discovered many other reasonable suggestions not included in our gold structure, such as two responses about "passenger vans". Additionally, though our gold structure only includes a 2-level hierarchy, in reality there is much more inherent structure (subcategories, etc.) that the model was likely suggesting.

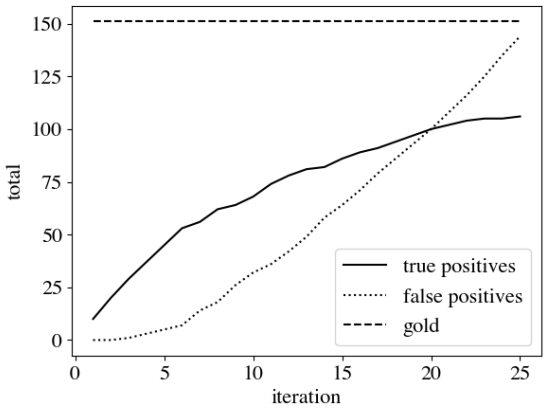

Figure 5: Results from virtual assistant case study. With 250 responses from the user we are able to recover 70% of the gold structure.

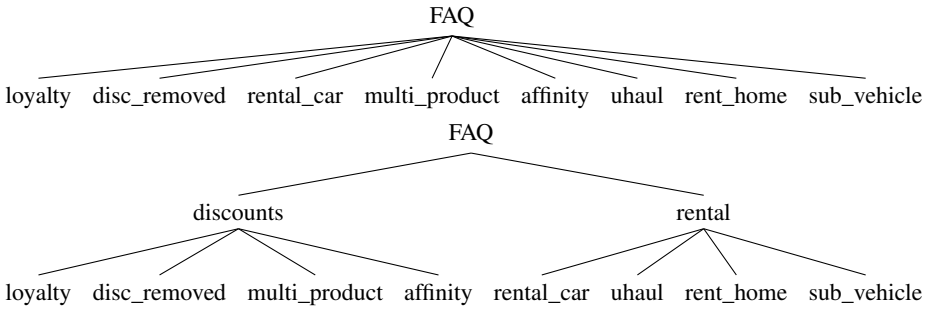

Figure 6: Example structure recovered by our procedure

6 Conclusion and Future Work

We have proposed a system for designing and driving conversational agents using a knowledge graph. Our aim is to better understand these systems and better enable them to scale out and succeed by formalizing the underlying processes. We also propose to use machine learning and human-in-the-loop procedures to design and curate the underlying knowledge graph to maximize their effectiveness. We specified one such approach and presented two case studies applying our systems to real applications in the insurance industry.

This area is very active for us and we are pursuing many directions of research and development. As mentioned so far in the paper, we would like to expand the knowledge graph restructuring process to include splits and other more complicated operations, as well as investigate the impact of restructuring on construction episodes. Furthermore, while our process proposes merges, it is still incumbent on the curator to determine properties or disambiguation strategies for the different subtypes under a new type; we would like to leverage machine learning techniques to suggest and create these properties to relieve more burden from the curators. We would also like to support multiple inheritance to allow more flexibility with the types we can express in our system.

Another common problem and area we'd like to explore is the issue of introducing new types or instances to a KGCA. Often it is the case that new types have little training data when compared to the types being used in production; it is essentially a few-short learning problem. We would like to see how the knowledge graph structure can improve performance for this case.

One benefit we get for using this system in production is the feedback we are able to obtain; we can analyze data to find patterns and use the frequency of certain types or instances as priors. For example, if a customer said they hit a deer, with enough data we hypothesize we would be able to determine that it is likely that their front bumper is damaged.

References

[1] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on freebase from question-answer pairs. In *EMNLP*, pages 1533–1544. ACL, 2013.

[2] Joseph Bockhorst, Devin Conathan, and Glenn M. Fung. Probabilistic-logic bots for efficient evaluation of business rules using conversational interfaces. In *Proceedings of the Thirty-third Conference on Innovative Applications of Artificial Intelligence, Jan 27-Feb 1 2019, Honolulu, Hawaii*, 2019.

[3] Arash Einolghozati, Panupong Pasupat, Sonal Gupta, Rushin Shah, Mrinal Mohit, Mike Lewis, and Luke Zettlemoyer. Improving semantic parsing for task oriented dialog. *CoRR*, abs/1902.06000, 2019.

[4] Sonal Gupta, Rushin Shah, Mrinal Mohit, Anuj Kumar, and Mike Lewis. Semantic parsing for task oriented dialog using hierarchical representations. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018.

[5] Mohit Iyyer, Wen-tau Yih, and Ming-Wei Chang. Search-based neural structured learning for sequential question answering. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 1821–1831, 2017.

[6] Yoon Kim. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014.

[7] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[8] Yibo Sun, Duyu Tang, Nan Duan, Jingjing Xu, Xiaocheng Feng, and Bing Qin. Knowledge-aware conversational semantic parsing over web tables. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence*, 2019.

[9] Yushi Wang, Jonathan Berant, and Percy Liang. Building a semantic parser overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1332–1342, Beijing, China, July 2015. Association for Computational Linguistics.

[10] Ziyu Yao, Xiujun Li, Jianfeng Gao, Brian M. Sadler, and Huan Sun. Interactive semantic parsing for if-then recipes via hierarchical reinforcement learning. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019.*, pages 2547–2554, 2019.

Appendix A KGCA

In this section we describe how to use the episode grammar to drive a conversational agent.

A.1 Prompts and utterances

We call message from the system to the user **prompts** and messages from the user to the system **utterances**. Our formulation does not impose constraints on the nature of either prompts or utterances. We expect there to be wide variety of forms that are dictated by the applications. Prompts range from plain text, rich text, discrete selectors such as sliders, buttons, check-boxes, constrained values (e.g., a number between 1 and 10), images, audio or video. Utterance formats are the complementary forms: text, selections and values.

A.2 Algorithm

Algorithm 2 High level pseudocode of a conversational episode

function KGCA($\mathcal{T}, \theta$)
  $\mathcal{T}$: A partial parse tree
  θ: the system's short-term memory
  if IsComplete(T) **then return** $\mathcal{T}$
  end if
  $u \leftarrow$ Turn($\mathcal{T}$)
  $\theta \leftarrow$ Update(θ, Match($u, \mathcal{T}$))
  $\mathcal{T} \leftarrow$ Expand($\mathcal{T}, \theta$)
  return KGCA($\mathcal{T}, \theta$)
end function

At the beginning of an episode the partial parse tree $\mathcal{T}$ consists of single node containing the nonterminal for the root type and the system's short-term memory θ is empty. As the episode progresses $\mathcal{T}$ grows on each turn. Episode termination is checked in IsComplete(). The termination condition for construction episodes is when we have a complete parse tree with no nonterminal leaves. For retrieval episodes the episode ends when $\mathcal{T}$ is consistent with only a single one of the parse trees of the candidate instances.

As shown in Algorithm 2 there are several key functions invoked in each step of the conversation. Turn represents one turn of the conversation in which the system and the user exchange messages. In the simplest case the system sends the user a single prompt to which the user responds with an utterance. Turn takes in $\mathcal{T}$ so that that the system can tailor the prompt according the information it needs to complete the parse. Match processes the user utterance in the context of the current parse and returns a encoding that is incorporated into the memory in Update. The exact processing of Match is context dependent. For example, when the prompt offers the user a Yes / No choice Match is very simple while a prompt for unstructured text such as "Please describe what happened" in the FNOL application Match involves applying a suite of entity recognizers and/or document classification models to the utterance. Lastly, Expand takes the updated memory and returns an expanded tree. Expand depends on the prompt (from Turn) and both are related to the conversational strategy.

A.3 Conversation Strategy

The main loop of KGCAs shown in Algorithm 2 gives a high-level overview. The details lie within the functions. The implementations of the key functions are closely related to the conversational strategy at play at the time. We describe two general conversational strategies here, but we note that i) there are many other variations of strategies possible and ii) the system may switch between the strategies during that course of an episode.

A.3.1 Fixed Increment

A fixed increment strategy is one that grows T by expanding one leaf nonterminal N per turn. Methods for choosing the nonterminal include information gain [2] and a predefined priority. After

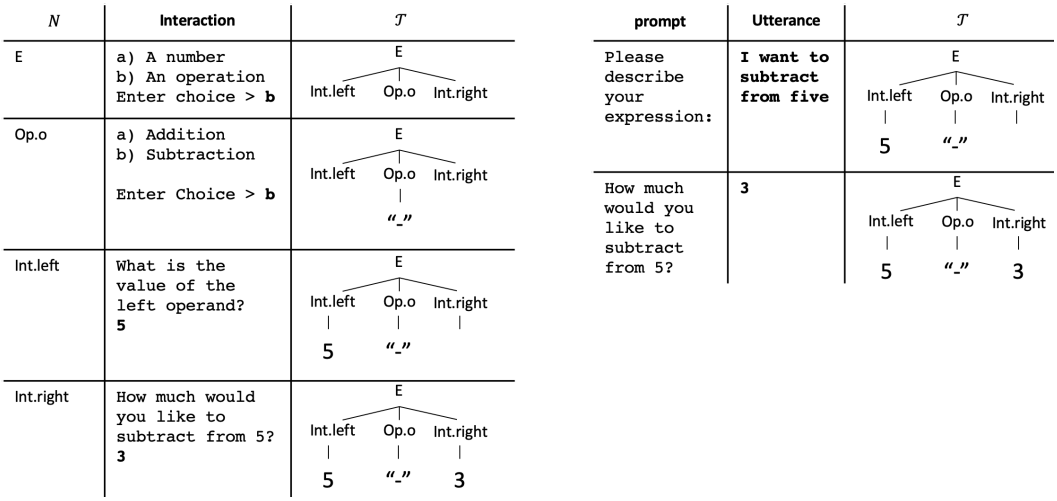

Figure 7: Example progressions of conversations using fixed-increment (left) and variable-increment (right) strategies. At the beginning of the episode the tree $\mathcal{T}$ is just the root node E (not shown). Each row shows a single turn where the tree $\mathcal{T}$ shown is the resulting one after `Expand`. For fixed-increment, N (first column) is the nonterminal selected to expand.

the system chooses N it sends a prompt to the user designed to solicit a response utterance indicating one of N's productions. `Match` interprets the utterance in this context mapping it to the selected production, which is subsequently stored in the memory θ (under pure single-step θ is otherwise empty). Finally, `Expand` extends $\mathcal{T}$ by appending the selected production.

A.4 Variable Increment

Under a variable increment strategy $\mathcal{T}$ grows by a arbitrary amount per-turn. Unlike a fixed strategy in which the prompt is narrowly focused on a single nonterminal, in this case the prompt is designed to solicit an utterance that more broadly indicates how to extend the tree. For instance, a prompt near the beginning of the FNOL agent instructs the user: "`Please describe what happened`". Our matching approach under variable increment scans the utterance against a collection of recognizers, each associated with a specific nonterminal in the EG. In general we use two classes of recognizers: static rules based on regular expressions and statistical models learned with machine learning approaches akin to methods for named entity recognition.

Appendix B Example: Arithmetic Expression

Figure 7 shows the evolution of $\mathcal{T}$ through episodes involving a fixed increment and variable increment strategy respectively on a toy example related to an integer expression. In this example we assume the KG contains a type hierarchy for expressions with root type `E` that gives rise to an episode grammar

`E` $\rightarrow$ `Int`
`E` $\rightarrow$ `Int.left Op.o Int.right`
`Op.o` $\rightarrow$ "+" | "-"

and productions for `Int`, `Int.left` and `Int.right` to integer literals. This grammar represents integer expressions that consist of either an integer literal or the addition or subtraction of two integer literals.

From these example we see instances of the following general concepts

- Fixed increment is simple but takes more turns
- Variable increment is more complex but may take fewer turns
- The prompt depends on all of $\mathcal{T}$ not just the selected nonterminal enabling more informative prompts such as in step 4 of fixed increment.

- Variable increment permits **production inference**. For example the first step infers the production `E` $\rightarrow$ `Int.left Op.o` because that production is necessary to produce "-" which was matched in the utterance.
- Nonterminal are allowed custom recognizers and are not restricted to be just a composition of recognizers for its 'parts'.
- The episode grammar does not presume any sequential constraints among the manifestations of nonterminal entities within user utterances. For example, in the first step of the variable increment case after the user says "I want to subtract from five" the system is free to link the value "5" with the left operand even though the operator "subtract" appears prior to "five" in the user utterance.