GraphQ IR: Unifying the Semantic Parsing of Graph Query Languages with One Intermediate Representation

Lunyiu Nie1, Shulin Cao1, Jiaxin Shi2, Jiuding Sun1, Qi Tian2, Lei Hou1, Juanzi Li1, Jidong Zhai1

1 Department of Computer Science and Technology, Tsinghua University
2 Huawei Cloud Computing Technologies Co., Ltd.

{nlx20, caosl19, sjd22}@mails.tsinghua.edu.cn, shijx12@gmail.com, tian.qi1@huawei.com, {houlei, lijuanzi, zhaijidong}@tsinghua.edu.cn

Abstract

Subject to the huge semantic gap between natural and formal languages, neural semantic parsing is typically bottlenecked by its complexity of dealing with both input semantics and output syntax. Recent works have proposed several forms of supplementary supervision but none is generalized across multiple formal languages. This paper proposes a unified intermediate representation (IR) for graph query languages, named GraphQ IR. It has a natural-language-like expression that bridges the semantic gap and formally defined syntax that maintains the graph structure. Therefore, a neural semantic parser can more precisely convert user queries into GraphQ IR, which can be later losslessly compiled into various downstream graph query languages. Extensive experiments on several benchmarks including KQA PRO, OVERNIGHT, GRAILQA and METAQA-Cypher under standard i.i.d., out-of-distribution and low-resource settings validate GraphQ IR’s superiority over the previous state-of-the-arts with a maximum 11% accuracy improvement.

1 Introduction

By mapping natural language utterances to logical forms, the task of semantic parsing has been widely explored in various applications, including database query (Yu et al., 2018; Talmor and Berant, 2018) and general-purpose code generation (Yin and Neubig, 2017; Campagna et al., 2019; Nan et al., 2020). Although the methodology has evolved from earlier statistical approaches (Zettlemoyer and Collins, 2005; Kwiatkowski et al., 2010) to present Seq2Seq paradigm (Zhong et al., 2017; Damonte and Monti, 2021), the semantic gap between natural language and logical forms still lies as the major challenge for semantic parsing.

As shown in Figure 1, in graph query languages (e.g., SPARQL, Cypher, Lambda-DCS, and newly emerged KoPL, etc.), graph nodes, edges and their respective properties constitute the key semantics of the logical forms (Pérez et al., 2009), which are very different from the expression of natural language utterances. Such discrepancy significantly hinders the learning of neural semantic parsers and therefore increases the demand for labeled data (Yin et al., 2022). However, due to the laborious efforts and language-specific expertise required in annotation, such demand cannot always be satisfied and thus becomes the bottleneck (Li et al., 2020b; Herzig et al., 2021).

To overcome these challenges, many works adopt complementary forms of supervision, such as the schema of database (Hwang et al., 2019), results of the execution (Clarke et al., 2010; Wang et al., 2018, 2021), and grammar-constrained decoding algorithms (Krishnamurthy et al., 2017; Shin et al., 2021; Baranowski and Hochgeschwender, 2021). Although effective, the additional resources that these methods rely on are not necessarily available in practice. By normalizing the expression (Berant and Liang, 2014; Su and Yan, 2017) or enriching the structure (Reddy et al., 2016; Cheng et al., 2017; Hu et al., 2018) of natural language utterances, another category of works proposes various intermediate representations like AMR (Kapanipathi et al., 2021) to ease the parsing of complex queries. However, the transition from their IRs to the downstream logical forms may incur extra losses in precision (Bornea et al., 2021). Besides, these representations are usually coupled to specific data or logical forms and thus cannot be easily transferred to other tasks or languages (Kamath and Das, 2019).

In industry, aside from SPARQL, many other graph query languages such as Cypher (Francis et al., 2018) and Gremlin (Rodriguez, 2015) are equally or even more commonly used in graph database interaction (Angles, 2012; Seifer et al., 2019). However, most graph query semantic parsing works only support SPARQL (Talmor and Berant, 2018; Dubey et al., 2019; Keysers et al., 2020)
Figure 1: A property graph extracted from Wikidata (Vrandecic and Krötzsch, 2014). We present a relevant user query with its corresponding logical forms in different query languages and in GraphQ IR.

while very few works target other graph query languages. Meanwhile, no existing tools or IR can support the data conversion among multiple graph query languages (Moreira and Ramalho, 2020; Agrawal et al., 2022). Such lack of interoperability has not only hindered the semantic parsing of low-resource languages but also limited the potential of querying heterogeneous databases (Mami et al., 2019; Angles et al., 2019).

In this paper, we propose a unified intermediate representation for graph query languages, namely GraphQ IR, to resolve these issues from a novel perspective. The designs of GraphQ IR weigh up the semantics of both natural and formal language by (a) producing the IR sequences with composition rules consistent with modern English (Tomlin, 2014) to close the semantic gap; and (b) maintaining the fundamental graph structures like nodes, edges, and properties, such that the IR can be automatically compiled into any downstream graph query languages without any loss.

Instead of directly mapping the user query to the logical form, we first parse natural language into GraphQ IR, then compile the IR into the target graph query languages (e.g., SPARQL, Cypher, Lambda-DCS, KoPL, etc.). Therefore, language-specific grammar features that initially posed a huge obstacle to semantic parsing are now explicitly handled by the compiler. Additionally, with the GraphQ IR as a bridge, our implemented source-to-source compiler can support lossless translation among multiple graph query languages and thus unify the annotations of different languages for eliminating the data bottleneck.

To validate the effectiveness of GraphQ IR, we conducted extensive experiments on benchmarks KQA PRO, OVERNIGHT, GRAIL.QA and METAQA-Cypher. Results show that our approach can consistently outperform the previous works by a significant margin. Especially under the compositional and few-shot generalization settings, our approach with GraphQ IR can demonstrate a maximum 11% increase in accuracy over the baselines.

The main contributions of our work include:

- We propose GraphQ IR for unifying the semantic parsing of graph query languages and present the IR design principles that are critical for bridging the semantic gap;
- Experimental results show that our approach can consistently achieve state-of-the-art performance across multiple benchmarks under the standard i.i.d, out-of-distribution, and low-resource settings.
- Our implemented source-to-source compiler unlocks data interoperability by supporting the bi-directional translation among different graph query languages. The code and toolkit are publicly available at https://github.com/Flitternie/GraphQ_IR.

2 GraphQ IR

In this section, we formalize the grammar and the expressiveness of our GraphQ IR based on the definition of property graph and regular path query. Then we summarize the design principles of GraphQ IR for bridging the semantic gap between natural and formal language as well as unifying different graph query languages.

2.1 Definition

As the top of Figure 1 demonstrates, a graph database can be expressed as a collection of property graphs that include Entity (graph nodes, e.g.,
Stanley Kubrick), Attribute (node properties, e.g., date of birth), Concept (node label, e.g., film), Relationship (graph edges, e.g., spouse) and Qualifier (edge properties, e.g., start time).

Therefore, to evaluate the expressiveness of GraphQ IR, we start by giving the definition of property graph: a directed labeled multigraph where each node or edge can contain a set of property-value pairs (Angles, 2018).

Definition 1 (Property graph). A property graph \( G \) is a tuple \((N, E, \rho, \lambda, \sigma)\) where:

1. \( N \) is a finite set of nodes.
2. \( E \) is a finite set of edges such that \( N \cap E = \emptyset \).
3. \( \rho : E \rightarrow (N \times N) \) is a total function. Specifically, \( \rho(e) = (n_1, n_2) \) refers \( e \) is a directed edge from node \( n_1 \) to \( n_2 \).
4. \( \lambda : (N \cup E) \rightarrow L \) is a partial function where \( L \) is a set of labels. Specifically, if \( \lambda(n) = l \) then \( l \) is the label of node \( n \).
5. \( \sigma : (N \cup E) \times P \rightarrow V \) is a partial function with \( P \) a set of properties and \( V \) a set of values. Specifically, if \( \sigma(n, p) = v \) then the property \( p \) of node \( n \) has value \( v \).

Subsequently, a graph path can be expressed as \( \pi = (n_1, e_1, ..., e_{k-1}, n_k) \) where \( k \geq 1 \) with each \( e_i \) being the edge between \( n_i \) and \( n_{i+1} \). The spelling of path, denoted as \( \lambda(\pi) \), is the concatenation of edge labels \( \lambda(e_1)...\lambda(e_{k-1}) \) (Mendelzon and Wood, 1995; Baeza, 2013).

Definition 2 (Regular path query). A regular path query has the general form \( Q = x \xrightarrow{\alpha} y \) where \( x \) denotes the start point, \( \alpha \) is a regular expression defined over \( \lambda(\pi) \), and \( y \) denotes the endpoints of the query.

By incorporating \( \rho, \lambda, \sigma \) and their inverse function \( \rho^{-1}, \lambda^{-1}, \sigma^{-1} \), such regular path query can be extended to support navigational queries towards any graph elements \( \psi \in (N \cup E \cup L \cup P \cup V) \) (Wood, 2012; van Rest et al., 2016). We can now evaluate the expressiveness of a language.

Definition 3 (Path query expressiveness). A path query \( q \) is expressible in a language \( \mathcal{L} \), if there exists an expression \( \varepsilon \in \mathcal{L} \) such that, for any subgraph \( G' \subseteq G \), we have \( \varepsilon(G') = q(G') \) (Fletcher et al., 2015).

We formalize GraphQ IR as a context-free grammar \((\mathcal{V}, \mathcal{S}, \mathcal{P})\) and present its non-terminals and productions in Appendix Table 7. Its \( \mathcal{V} \) and \( \mathcal{P} \) are respectively defined as the superset of the terminal set \((n, e, l, p, v)\) and production set \((\rho, \lambda, \sigma, \rho^{-1}, \lambda^{-1}, \sigma^{-1})\) of regular graph query. Therefore, all path queries expressible in regular grammar are also expressible in the context-free grammar of GraphQ IR (Hopcroft et al., 2007). Furthermore, GraphQ IR also supports extended operations like Union, Difference and Filter to express complex graph query patterns (Angles et al., 2017).

Empirically, GraphQ IR can express all graph query patterns that appeared in benchmarks KQA PRO, OVERNIGHT, GRAILQA and METATHY, with details elaborated in Section 4.1.

2.2 Principles

We summarize several principles in designing GraphQ IR in this way: present in a syntax close to natural language while preserving the structural semantics equivalent to formal languages.

2.2.1 Diminishing syntactical discrepancy

To facilitate the training of the neural semantic parser, the target IR sequence should share a similar syntax in correspondence to the input utterance.

To achieve this, the IR structure should first match how users typically raise queries. Therefore, we simplify the triple-based structure in graph query languages into a more natural subject-verb-object syntactic construction (Tomlin, 2014). Take Figure 1’s task setting as an example, the two triples \( (?e \text{ instance_of } ?c) \) and \( (?c \text{ name } "film") \) as the entity concept constraint in SPARQL are simplified to the sentence subject “\(<\text{c} \text{ film } </\text{c}>\)” in GraphQ IR. Multi-hop relationship and attribute queries are formulated as relative clauses similar to the English expression and thus can be comfortably generated by a language-model-based neural semantic parser.

Secondly, IR should also leave out the variables (e.g., \( ?e, ?c \) in SPARQL) and operators (e.g., SELECT, WHERE, RETURN, etc.) in logical forms that cannot be easily aligned to natural language utterances. Alternatively, human-readable operators are adopted in GraphQ IR, as illustrated in Appendix Table 7.

2.2.2 Eliminating semantic ambiguity

In formal languages, multiple parallel implementations can achieve the same functionalities. However, such redundancy and ambiguity in semantics may pose challenges to the neural semantic parser.

For example, in Lambda-DCS, there co-exist three implementations for constraining one’s concept (e.g., Kobe is a player), respectively through:
When designing an IR, such redundant and ambiguous semantics should be clarified into more definitive and orthogonal representations (Campaagna et al., 2019). Thus in GraphQ IR, we unify all such unnecessary distinctions and prune redundant structures in logical forms to distill the core semantics. In the previous example, GraphQ IR only requires a simple noun modifier “<C> player</C>” as the concept constraint. This not only makes the language clearer for users and semantic parsers to comprehend, but also facilitates the next-step compilation from the IR to the downstream formal language.

2.2.3 Maintaining graph structural semantics

In addition to the aforementioned designs to improve alignment with natural language, the syntax of IR also needs to maintain the key structures of graph queries for subsequent lossless compilation. Specifically, IR should keep track of the data types of graph structural elements. We design GraphQ IR to be strong-typing by explicitly stating the type of terminal nodes with respective special tokens, e.g., <E> for Entity, <R> for Relation, <A> for Attribute, etc. Values of different types are also differentiated in GraphQ IR with our pre-defined or user custom indicators, e.g., string, number, date, time, etc.

Furthermore, IR should also preserve the hierarchical dependencies that are critical for multi-hop queries. We introduce <ES> as a scoping token in GraphQ IR to explicitly indicate the underlying dependencies among the clauses produced by an EntitySet, as shown in Appendix Table 7. Such scoping tokens in GraphQ IR can facilitate the compiler to recover the hierarchical structure and finally convert the IR sequences into one of the graph query languages deterministically.

3 Implementation

We depict the full picture of our proposed framework in Figure 2. The neural semantic parser first maps the input natural language utterance into GraphQ IR. Thereafter, the GraphQ IR sequence is fed into the compiler and parsed into an abstract syntax tree for downstream graph query language code generation.

3.1 Neural Semantic Parser

To verify the above principles in practice, we formulate the conversion from natural language to our GraphQ IR as a Seq2Seq task and adopt an encoder-decoder framework for implementing the neural semantic parser.

As shown in the left part of Figure 2, the encoder module of the semantic parser first maps the input natural language utterance X to a high dimensional feature space with non-linear transformations for capturing the semantics of the input tokens. The decoder module subsequently then interprets the hidden representations and generates the IR sequence.

Figure 2: Overall implementation of our proposed framework. The user queries are first converted to GraphQ IR sequences by a semantic parser and subsequently transpiled into the target graph query languages by a compiler.
by factorizing the probability distribution:

$$p(\text{IR}) = \prod_{i=1}^{n} P(y_i|X, y_1, ..., y_{i-1}),$$  \hspace{1cm} (1)$$

where $y_i$ is the $i$-th token of IR sequence with in total $n$ tokens. Specifically, we implement this encoder-decoder network with BART (Lewis et al., 2020), a pretrained language model that is proficient in comprehending the diverse user utterances and generating the GraphQ IR sequences that are structured in natural-language-like expressions.

Please note that the implementation in this part is orthogonal to our GraphQ IR and can be substituted by other semantic parsing models.

### 3.2 Compiler

The implementation of GraphQ IR’s compiler comprises a front-end module that generates an abstract syntax tree from the IR sequence and a back-end module that transforms the tree structure into the target graph query language.

The compiler front-end is responsible for performing the lexical and syntax analysis on the IR sequence. The lexer first splits the sequence into lexical tokens, which are subsequently structured into a parse tree with LL(*') parsing strategy (Parr, 2013) according to the pre-defined grammar in Section 2.1. As such, GraphQ IR sequence can be automatically constructed into an abstract syntax tree (AST) that contains syntactic dependencies and hierarchical structures.

The compiler back-end will then traverse the abstract syntax tree and restructure the nodes and dependencies into one of the downstream graph query languages. We formalize the code generation as a tree mapping process, where the subtrees carrying equivalent information are aligned according to pre-defined transformation rules. To illustrate, we present 2 examples of generating SPARQL and Lambda-DCS queries respectively in Appendix Figure 5 and Figure 4.

Similarly, we also implement the compiler that supports conversion from graph query languages to GraphQ IR. Thus, with the IR as a middleware, our toolkit can also achieve the transpilation between any two graph query languages supported.

### 4 Experiments

In this section, we evaluate GraphQ IR on several benchmarks under different task settings.

### 4.1 Datasets

For evaluation, we test on benchmarks KQA PRO, OVERNIGHT, GRAILQA and METAQA-Cypher that altogether cover graph query languages SPARQL, KoPL, Lambda-DCS, and Cypher.

In all experiments, the GraphQ IR sequences are automatically converted from the original logical forms of the respective datasets by the bidirectional compiler without extra re-annotation.

**KQA PRO** KQA PRO (Cao et al., 2022a) is a large-scale dataset for complex question answering over Wikidata knowledge base (Vrandecic and Krötzsch, 2014). It is the largest KBQA corpus that contains 117,790 natural language questions along with the corresponding SPARQL and KoPL logical forms, covering complex graph queries involving multi-hop inference, logical union and intersection, etc. In our experiment, it is divided into 94,376 train, 11,797 validation, and 11,797 test cases.

**Overnight** OVERNIGHT (Wang et al., 2015) is a semantic parsing dataset with 13,682 examples across 8 sub-domains extracted from Freebase (Bollacker et al., 2008). Each domain has natural language questions and pairwise Lambda-DCS queries executable on SEMPRE (Berant et al., 2013). It exhibits diverse linguistic phenomena and semantic structures across domains, e.g., temporal knowledge in CALENDAR domain and spatial knowledge in BLOCKS domain. We use the same train/val/test splits as in the previous work (Wang et al., 2015).

**GRAILQA** GRAILQA (Gu et al., 2021) is a knowledge base question answering dataset with 64k questions grounded on Freebase (Bollacker et al., 2008) that evaluate generalizability at three levels, i.e., i.i.d, compositional generalization and zero-shot. To focus on the sole task of semantic parsing, we replace the entity IDs (e.g., m.06mn7) with their respective names (e.g., Stanley Kubrick) in GRAILQA’s logical forms, thus eliminating the need for an explicit entity linking module as in previous works (Chen et al., 2021; Ye et al., 2022). Since GRAILQA’s test set is not publicly available for such transformation, we report the validation set results for our evaluation, which have been studied to show consistent trends with the test set (Gu and Su, 2022).

**METAQA-Cypher** METAQA (Zhang et al., 2018) contains more than 400k multi-hop QA pairs over WikiMovies knowledge base (Miller et al.,
### Baselines

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RGCN (Schlichtkrull et al., 2018)</td>
<td>34.00 27.61 30.03 35.85 41.91 65.88</td>
</tr>
<tr>
<td>BART+SPARQL (Cao et al., 2022a)</td>
<td>88.49 83.09 96.12 88.67 85.78 92.33</td>
</tr>
<tr>
<td>BART+KoPL (Cao et al., 2022a)</td>
<td>89.46 84.76 95.51 89.30 86.68 93.30</td>
</tr>
<tr>
<td>CFQ IR (Herzig et al., 2021)</td>
<td>87.51 81.32 95.70 90.33 86.23 92.20</td>
</tr>
</tbody>
</table>

### Our Approach

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphQ IR</td>
<td>90.38 84.90 97.15 92.64 89.39 94.20</td>
</tr>
</tbody>
</table>

### Table 1: Test accuracies on KQA PRO dataset. Data are categorized into Multi-hop queries with multi-hop inference, Qualifier knowledge queries, Comparison between several entities, Logical union or intersection, Count queries for the quantity of entities, Verify queries with a boolean answer, and Zero-shot queries whose answer is not seen in the training set.

### Baselines

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SPO (Wang et al., 2015)</td>
<td>46.3 41.9 74.4 54.0 59.0 70.8 75.9 48.2 58.8</td>
</tr>
<tr>
<td>CrossDomain* (Su and Yan, 2017)</td>
<td>88.2 62.2 82.1 78.8 80.1 86.1 83.7 83.1 80.6</td>
</tr>
<tr>
<td>Seq2Action (Chen et al., 2018a)</td>
<td>88.2 61.4 81.5 74.1 80.7 82.9 80.7 82.1 79.0</td>
</tr>
<tr>
<td>DUAL (Cao et al., 2019)</td>
<td>84.9 61.2 78.6 67.2 78.3 80.6 78.9 81.3 76.4</td>
</tr>
<tr>
<td>2-stage DUAL* (Cao et al., 2020)</td>
<td>87.2 65.7 80.4 75.7 80.1 86.1 82.8 82.7 80.1</td>
</tr>
</tbody>
</table>

### Our Approach

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphQ IR</td>
<td>88.2 64.7 78.6 72.0 77.6 83.3 84.9 81.6 79.5</td>
</tr>
<tr>
<td>GraphQ IR*</td>
<td>88.2 65.4 81.6 81.5 82.6 92.9 89.8 84.1 82.1</td>
</tr>
</tbody>
</table>

### Table 2: Test accuracies on OVERNIGHT dataset. Methods with asterisk (*) involve cross-domain training.

### 4.2 Metric

We adopt execution accuracy as our metric based on whether the generated logical form queries can return correct answers. For queries with multiple legal answers, we require the execution results to exactly match all ground-truth answers.

### 4.3 Results

#### I.I.D. Generalization

As Table 1 illustrates, on KQA PRO, our proposed approach with GraphQ IR consistently outperforms the previous approaches on all query categories. In particular, GraphQ IR exhibits good generalization under the complex Multi-hop, Qualifier and Zero-shot settings with even larger margins over the baselines. We attribute this to its natural-language-like representations that effectively close the semantic gap and its formally-defined syntax that can be losslessly converted into downstream languages.

As for OVERNIGHT, our methods also significantly surpass the baselines as shown in Table 2.

Previous works usually train separate parsers for each of the eight domains due to their distinct vocabularies and grammars (Wang et al., 2015; Chen et al., 2018a). With an extra layer of GraphQ IR for unification, domain-specific data are now consolidated into one universal representation, and the training of one domain can thereby benefit from the others. Consequently, GraphQ IR* that gets trained on the aggregate data of all eight domains demonstrates the best results.

#### OOD Generalization

Current neural semantic parsers often fail in generalizing to out-of-distribution (OOD) data (Pasupat and Liang, 2015; Keysers et al., 2020; Furrer et al., 2020). Therefore, we experiment on GAILQA, a dataset that...
specifying non-i.i.d. generalization.

We present the results in Table 3. Among the models without explicit entity linking modules, compared with the BART baseline that directly maps to the logical forms and the CFQ IR (Herzig et al., 2021) that particularly aims at SPARQL compositional generalization, GraphQ IR achieves the best overall performance and performs remarkably well also in compositional generalization and zero-shot data splits. This can be credited to our IR designs that clarify the redundant semantics and maintain the key hierarchical structure where its components can be flexibly combined or decomposed according to the pre-defined production rules.

**Low-resource Generalization** To verify whether GraphQ IR can aid the semantic parsing of low-resource languages, we reconstruct the METAQA dataset into Cypher, a graph query language commonly used in the industry but rarely studied in previous semantic parsing works (Seifer et al., 2019). To simulate the low-resource scenario, we adjust the data split to ensure that only 1, 3, and 5 samples of each question type appear in the training set under the 1-, 3-, and 5-shot settings.

The results in Table 4 indicate that our methods can remain robust under a low-resource setting with strong few-shot generalization. Specifically, the GraphQ IR* model that has in advance trained on KQA PRO (a dataset annotated in SPARQL and KoPL) demonstrates the most outstanding performance on METAQA-Cypher, especially under the most challenging 1-shot setting. Previous works in semantic parsing usually target a specified type of logical form and neglect the data interoperability across languages. With the GraphQ IR as a bridge, low-resource query languages can now leverage data from other languages. A universal semantic parser that can end-to-end support different languages also becomes possible.

5 Discussion

To further explore the reasons behind the superior performance of our methods, we compute and visualize the semantic distance between the natural language utterances and their corresponding logical forms.

Specifically, to simulate how a neural semantic parser processes the sequences in the above experiments, we use a pretrained BART-base model without fine-tuning to obtain the contextualized embeddings (Li et al., 2020a). For each sequence, we...
Table 6: The analysis of 4 error types based on the failure cases as occurred in benchmark KQA PRO’s test data. “# OSC” refers to the number of errors that can be fixed with one step correction on the IR’s structure.

<table>
<thead>
<tr>
<th>Error Type</th>
<th>#</th>
<th>OSC</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inaccurate data annotation</td>
<td>28</td>
<td>-</td>
<td>User utterance: Out of newscasts that last 110 minutes, which is the shortest?</td>
</tr>
<tr>
<td>Generated IR</td>
<td>13</td>
<td>-</td>
<td>when incorrect logical forms are generated. Out of 1000 queries respectively from KQA PRO and OVERNIGHT’s validation set. We randomly sampled 1000 cases and categorized them into 4 types as shown in Table 6.</td>
</tr>
<tr>
<td>GraphQ IR and semantic parser.</td>
<td></td>
<td></td>
<td>When did Joseph L. Mankiewicz graduate from Columbia University?</td>
</tr>
<tr>
<td>Generated IR</td>
<td>27</td>
<td>27</td>
<td>when incorrect logical forms are generated. Out of 1000 queries respectively from KQA PRO and OVERNIGHT’s validation set. We randomly sampled 1000 cases and categorized them into 4 types as shown in Table 6.</td>
</tr>
<tr>
<td>Unspecified graph structure</td>
<td>13</td>
<td>9</td>
<td>when incorrect logical forms are generated. Out of 1000 queries respectively from KQA PRO and OVERNIGHT’s validation set. We randomly sampled 1000 cases and categorized them into 4 types as shown in Table 6.</td>
</tr>
<tr>
<td>Nonequivalent semantics</td>
<td>32</td>
<td>25</td>
<td>when incorrect logical forms are generated. Out of 1000 queries respectively from KQA PRO and OVERNIGHT’s validation set. We randomly sampled 1000 cases and categorized them into 4 types as shown in Table 6.</td>
</tr>
</tbody>
</table>

Table 6: The analysis of 4 error types based on the failure cases as occurred in benchmark KQA PRO’s test data. “# OSC” refers to the number of errors that can be fixed with one step correction on the IR’s structure.

5.1 Error Analysis

To investigate GraphQ IR’s potentials and bottlenecks, we look into the failures of our approach when incorrect logical forms are generated. Out of the total 979 errors in KQA PRO’s test set, we randomly sampled 100 cases and categorized them into 4 types as shown in Table 6.

5.1.1 Inaccurate data annotation (28%). The reference logical form (e.g., \( v_1 \neq "110" \)) may contain inconsistent or misinterpreted information that contradicts to the corresponding natural language utterance (e.g., last 110 minutes). We attribute this type of error to the dataset rather than the failure of our approach.

5.1.2 Ambiguous query expression (27%). The semantics of the user utterance may be present in more than one way (e.g., \( \text{kid film or children’s film} \)) due to the ambiguity in natural language, whereas the schema of the knowledge base is pre-defined (e.g., only children’s film is considered a valid entity). This category of error can be fixed by incorporating explicit schema linking modules, which are orthogonal to the implementation of our GraphQ IR and semantic parser.

5.1.3 Unspecified graph structure (13%). Logical forms of different structures (e.g., (Uzbekistan
capital Tashkent) and (Tashkent capital_of Uzbekistan)) can convey the same semantics in a directed cycle graph, but some of them contain structures that are absent in a knowledge base. This type of error is due to the incompleteness of the knowledge base.

Nonequivalent semantics (32%). The output includes incorrect query element (e.g., string and numerical values) or structure (e.g., edges and properties) that conveys nonequivalent semantics, such as misinterpreting graduate to start_time.

Overall, 89% of the sampled errors can be simply fixed by the revision of annotation or one-step correction on the IR element, demonstrating that our proposed method with GraphQ IR can generate high-quality logical forms that are easy to debug.

6 Related Work

6.1 Semantic Parsing

Semantic parsing aims to translate natural language utterances into executable logical forms, such as CCG (Zettlemoyer and Collins, 2005), Lambda-DCS (Liang, 2013; Pasupat and Liang, 2015), SQL (Zhong et al., 2017; Yu et al., 2021), AMR (Banarescu et al., 2013), SPARQL (Sun et al., 2020) and KoPL (Cao et al., 2022a,b).

Most recent works take semantic parsing as a Seq2Seq translation task via an encoder-decoder framework, which is challenging due to the semantic and structural gaps between natural utterances and logical forms. To overcome such issues, current semantic parsers usually (1) rely on a large amount of labeled data (Cao et al., 2022a); or (2) leverage external resources for mini the structural mismatch, e.g., injecting grammar rules during decoding (Wu et al., 2021; Shin et al., 2021); or (3) employ synthetic data to diminish the semantic mismatch (Xu et al., 2020; Wu et al., 2021).

Compared with previous works, our proposed GraphQ IR allows the semantic parser to adapt to different downstream formal query languages without extra efforts and demonstrates promising performance under the compositional generalization and few-shot settings.

6.2 Intermediate Representation

Intermediate representations (IR) are usually generated for the internal use of compilers and represent the code structure of input programs (Aho et al., 1986). Good IR designs with informative and distinctive mid-level features can provide huge benefits for optimization, translation, and downstream code generation (Lattner and Adve, 2004), especially in areas like deep learning (Chen et al., 2018b; Cyphers et al., 2018) and heterogeneous computing (Lattner et al., 2020).

Recently, IR has also become common in many semantic parsing works that include an auxiliary representation between natural language and logical form. Most of them take a top-down approach and adopt IR similar to natural language (Su and Yan, 2017; Herzig and Berant, 2019; Shin et al., 2021). In contrast, another category of works constructs IR based on the key structure of target logical forms in a bottom-up manner (Wolfson et al., 2020; Marion et al., 2021). For example, Herzig et al. designed CFQ IR that rewrites SPARQL by grouping the triples of identical elements (2021).

Although these works partially mitigate the mismatch between natural and formal language, they either failed in removing the formal representations that are unnatural to the language models or neglected the structural information requisite for downstream compilation. In this work, we omit those IRs that cannot be losslessly converted into downstream logical forms.

7 Conclusion and Future Work

This paper proposes a novel intermediate representation, namely GraphQ IR, for bridging the semantic gap between natural language and graph query languages. Evaluation results show that our approach with GraphQ IR consistently surpasses the baselines on several benchmarks covering multiple formal languages, i.e., SPARQL, KoPL, Lambda-DCS, and Cypher. Moreover, GraphQ IR also demonstrates superior generalization ability and robustness under the out-of-distribution and low-resource settings.

As an early step towards the unification of semantic parsing, our work opens up several future directions. For example, many code optimization techniques (e.g., common subexpression elimination) can be incorporated into IR to improve performance further. By bringing in multiple levels of IR, our framework may also be extended to support relational database query languages like SQL. Moreover, since the current designs of GraphQ IR still require non-trivial manual efforts, the automation of such procedure, e.g., in prompt-like manners, is worth future exploration.
Limitations

The major limitations of this work include: (a) the composition rules of GraphQ IR are closely aligned with interrogative sentences. Therefore, our current formalism may not be applicable to general-domain semantic parsing; (b) for the semantic parsing of an input language whose syntax significantly differs from English (e.g., Arabic, Chinese, Hindi, etc.), the benefits of GraphQ IR may be limited; (c) our experiments fine-tuned a neural semantic parser on top of a pretrained model with ~139 million parameters, thus cannot be easily reproduced without adequate GPU resources.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments. This work is partially supported by the National Key R&D Program of China (2021ZD0110104), the National Natural Science Foundation of China (U20A20226), the NSFC Youth Project (62006136), and a grant from the Institute for Guo Qiang, Tsinghua University (2019GQB0003). Jidong Zhai is the corresponding author of this paper.

References

Lakshya A. Agrawal, Nikunj Singhal, and Raghava Mutharaju. 2022. A SPARQL to cypher transpiler: Proposal and initial results. In CODS-COMAD 2022: 5th Joint International Conference on Data Science & Management of Data (9th ACM IKDD CODS and 27th COMAD), Bangalore, India, January 8 - 10, 2022, pages 312–313. ACM.


Giovanni Campagna, Silei Xu, Mehrad Moradshahi, Richard Socher, and Monica S. Lam. 2019. Genie: a generator of natural language semantic parsers for...


Richard Shin, Christopher H. Lin, Sam Thomson, Charles Chen, Subhro Roy, Emmanouil Antonios Platanios, Adam Pauls, Dan Klein, Jason Eisner, and...


Tao Yu, Chien-Sheng Wu, Xi Victoria Lin, Bailin Wang, Yi Chern Tan, Xinyi Yang, Dragomir R. Radev,
A GraphQ IR Grammar

We present GraphQ IR’s non-terminals and production rules in Table 7.

B Implementation Details

B.1 Model Hyperparameters

For the neural semantic parser, we used the BART-base model (Lewis et al., 2020) released by Facebook on HuggingFace1. 12 special tokens (e.g., <E>) were added to the tokenizer vocabulary as the structure indicators for GraphQ IR. We used the AdamW optimizer (Loshchilov and Hutter, 2017) with the learning rate set to $3 \times 10^{-5}$ and weight decay set to $1 \times 10^{-5}$ following the default settings.

B.2 Environmental Configurations

In our implementation of the compiler, we used ANTLR (Parr, 2013) version 4.9.2 for analyzing our specified grammar rules and building up the corresponding lexer and parser toolkit. For evaluation, we used Virtuoso 7.202, SEMPRE 2.43, Neo4j 4.44 and KoPL 0.35 as the back-ends respectively for executing the SPARQL, Lambda-DCS, Cypher, and KoPL queries.

Our whole experiments were performed on a single machine with 8 NVIDIA Tesla V100 (32GB memory) GPUs on CUDA 11.

C Supplementary Study

C.1 KQA PRO Compositional Generalization

Compositional generalization refers to a model’s capability of generalizing from the known components to produce novel combinations (Pasupat and Liang, 2015; Keysers et al., 2020; Furrer et al., 2020). To measure our IR’s compositional generalization ability, we also create a new KQA PRO data split based on the logical form length and test the parsers to generate long queries (KoPL queries with $>7$ functions) based on the short query components seen in the training data (KoPL queries with $\leq 7$ functions).

The results are listed in Table 8. Compared with the plain-BART baseline and the CFQ IR (Herzig et al., 2021) that is specially designed for improving the compositional generalization on SPARQL,

---

1https://huggingface.co/facebook/bart-base
2https://github.com/openlink/virtuoso-opensource
3https://github.com/percyliang/sempre
4https://github.com/neo4j/neo4j
5https://pypi.org/project/KoPL/
Table 7: GraphQ IR grammar rules that cover the common graph query patterns. “|” separates multiple productions at the same level, and “?” denotes that the preceding expression is optional. Italic words refer to the terminal symbols. Here we omit the corner case production rules for simplicity.

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>Productions</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>→ EntityQuery</td>
</tr>
<tr>
<td>EntityQuery</td>
<td>→ what is EntitySet</td>
</tr>
<tr>
<td>AttributeQuery</td>
<td>→ what is the attribute of EntitySet</td>
</tr>
<tr>
<td>RelationQuery</td>
<td>→ what is the relation from EntitySet to EntitySet</td>
</tr>
<tr>
<td>QualifierQuery</td>
<td>→ what is the qualifier of EntitySet Constraint</td>
</tr>
<tr>
<td>CountQuery</td>
<td>→ how many EntitySet</td>
</tr>
<tr>
<td>VerifyQuery</td>
<td>→ whether EntitySet Constraint</td>
</tr>
<tr>
<td>ValueQuery</td>
<td>→ what is Value</td>
</tr>
<tr>
<td>EntitySet</td>
<td>→ &lt;ES&gt; EntitySet LOP EntitySet &lt;/ES&gt;</td>
</tr>
<tr>
<td>Constraint</td>
<td>→ AttributeConstraint QualifierConstraint?</td>
</tr>
<tr>
<td>AttributeConstraint</td>
<td>→ whose Attribute COP Value that have SOP Attribute</td>
</tr>
<tr>
<td>RelationConstraint</td>
<td>→ that Relation DIR to (COP Value?) EntitySet</td>
</tr>
<tr>
<td>QualifierConstraint</td>
<td>→ Qualifier COP Value</td>
</tr>
<tr>
<td>Entity</td>
<td>→ &lt;E&gt; entity &lt;/E&gt;</td>
</tr>
<tr>
<td>Concept</td>
<td>→ &lt;C&gt; concept &lt;/C&gt;</td>
</tr>
<tr>
<td>Attribute</td>
<td>→ &lt;A&gt; attribute &lt;/A&gt;</td>
</tr>
<tr>
<td>Relation</td>
<td>→ &lt;R&gt; relation &lt;/R&gt;</td>
</tr>
<tr>
<td>Qualifier</td>
<td>→ &lt;Q&gt; qualifier &lt;/Q&gt;</td>
</tr>
<tr>
<td>Value</td>
<td>→ VTYPE &lt;V&gt; Value LOP Value &lt;/V&gt;</td>
</tr>
<tr>
<td>LOP</td>
<td>→ and</td>
</tr>
<tr>
<td>VOP</td>
<td>→ sum</td>
</tr>
<tr>
<td>SOP</td>
<td>→ is</td>
</tr>
<tr>
<td>DIR</td>
<td>→ forward</td>
</tr>
<tr>
<td>VTYPE</td>
<td>→ string</td>
</tr>
</tbody>
</table>

Table 8: Experimental results on KQA PRO compositional generalization data split.

<table>
<thead>
<tr>
<th></th>
<th>Overall</th>
<th>Qualifier</th>
<th>Comparison</th>
<th>Logical</th>
</tr>
</thead>
<tbody>
<tr>
<td>BART</td>
<td>50.58</td>
<td>21.55</td>
<td>87.66</td>
<td>50.60</td>
</tr>
<tr>
<td>CFQ</td>
<td>50.70</td>
<td>25.33</td>
<td>93.77</td>
<td>50.73</td>
</tr>
<tr>
<td>GraphQ IR</td>
<td><strong>54.91</strong></td>
<td><strong>40.46</strong></td>
<td><strong>95.19</strong></td>
<td><strong>54.90</strong></td>
</tr>
</tbody>
</table>

GraphQ IR achieves the best performance in overall data as well as in complex task settings, which can be again credited to our IR designs that simplify the redundant semantics and preserve the key structural features.
Natural Language Utterance:
friends of people who joined their jobs before 2005

GraphQ IR Sequence:
what is <ES> <ES> <C> person </C> </ES> that <R> friend </R> backward to <ES> <C> employee </C> <ES> ones whose <A> employment start date </A> at most year <V> 2004 </V> </ES> </ES> </ES>

GraphQ IR Abstract Syntax Tree:

Lambda-DCS Abstract Syntax Tree:

Lambda-DCS Sequence:
( call @listValue ( call @filter ( call @getProperty ( call @singleton en.person ) ( string ! type ) ) ( call @reverse ( string friend ) ) ( string = ) ( call @getProperty ( ( lambda s ( call @filter ( var s ) ( call @ensureNumericProperty ( string employment_start_date ) ) ( string <= ) ( call @ensureNumericEntity ( date 2004 -1 -1 ) ) ) ( call @domain ( string employee ) ) ) ( string employee ) ) ) )

Figure 4: A user query in OVERNIGHT. The neural semantic parser first converts the input utterance into GraphQ IR. The compiler then parses the GraphQ IR sequence into an abstract syntax tree, which is subsequently transformed into the corresponding Lambda-DCS sequence along with a tree mapping process. To exemplify, the subtrees circled by red dash lines are carrying equivalent information that can be transformed with pre-defined rules. The red words are terminal nodes that correspond to the graph structure.
Natural Language Utterance:
Which has less elevation above sea level, Rome that is the filming location of To Rome with Love or Lisbon which is the twinned administrative body of Santo Domingo?

GraphQ IR Sequence:
which one has the smallest <A> elevation above sea level </A> among <ES> <ES> <E> Rome </E> (<ES> ones that <R> filming location </R> backward to <E> To Rome with Love </E> </ES>) or <ES> <E> Lisbon </E> (<ES> ones that <R> twinned administrative body </R> backward to <E> Santo Domingo </E> </ES>) </ES> </ES>

GraphQ IR Abstract Syntax Tree:

SPARQL Abstract Syntax Tree:

SPARQL Sequence:
SELECT ?e WHERE { { ?e <pred:name> "Rome" . ?e_1 <filming_location> ?e . ?e_1 <pred:name> "To Rome with Love" . } UNION { ?e <pred:name> "Lisbon" . ?e_1 <twinned_administrative_body> ?e . ?e_1 <pred:name> "Santo Domingo" . } } ?e <elevation_above_sea_level> ?pv . ?pv <pred:value> ?v . } ORDER BY ?v LIMIT 1

Figure 5: A user query in KQA Pro. Similarly, the compiler parses the generated GraphQ IR sequence into an abstract syntax tree, then transforms its tree structure into the corresponding SPARQL sequence.