Abstract

Subject to the semantic gap lying between natural and formal language, neural semantic parsing is typically bottlenecked by the paucity and imbalance of data. In this paper, we propose a unified intermediate representation (IR) for graph query languages, namely GraphQ IR. With the IR’s natural-language-like representation that bridges the semantic gap and its formally defined syntax that maintains the graph structure, neural semantic parser can more effectively convert user queries into our GraphQ IR, which can be later automatically compiled into different downstream graph query languages. Extensive experiments show that our approach can consistently achieve state-of-the-art performance on benchmarks KQA PRO, OVERTNIGHT and METAQA. Evaluations under compositional generalization and few-shot learning settings also validate the promising generalization ability of GraphQ IR with at most 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 like querying database (Zhong et al., 2017; Yu et al., 2018) or knowledge base (Zhang et al., 2018; Talmor and Berant, 2018), virtual assistant conversation (Campagna et al., 2019; Fischer et al., 2021) and general-purpose code generation (Ling et al., 2016; Yin and Neubig, 2017; Nan et al., 2020). Early attempts in this field usually rely on compositional grammar such as CCG and DCS (Zettlemoyer and Collins, 2005; Kwiatkowksi et al., 2010; Liang et al., 2013). Recently, most works formulate semantic parsing as a Seq2Seq problem and adopt encoder-decoder neural networks that only require parallel data of natural language utterance and corresponding logical form for supervision (Zhong et al., 2017; Yu et al., 2020; Damonte and Monti, 2021).

However, these neural approaches still suffer from two major challenges: (a) Semantic gap. As shown in Figure 1, in graph query languages (e.g., SPARQL, Cypher, and newly emerged KoPL, etc.), graph nodes and edges constitute the key semantics of the logical forms (Pérez et al., 2009), which are very different from the expression of natural language queries. Such discrepancy significantly hinders the learning of neural semantic parser, especially in the era of pretrained language models (Lewis et al., 2020). (b) Imbalance of data. Due to the intensive labor and language-specific expertise required in annotation (Li et al., 2020), in spite of the various datasets prepared in SPARQL (Talmor and Berant, 2018; Dubey et al., 2019; Keysers et al., 2019), very few works target the semantic parsing of other graph query languages, such as Cypher and Gremlin, that are commonly used in industries (Seifer et al., 2019). Moreover, datasets of different languages are also isolated since no existing tools can support the conversion (Agrawal et al., 2022). Such imbalance and isolation of data has impeded the semantic parsing of low-resource languages in both academia and industry.

To overcome the above challenges, many works adopt complementary forms of supervision, such as the schema of database (Hwang et al., 2019), execution results of the logical forms (Clarke et al., 2010; Wang et al., 2018, 2021), constrained decoding algorithms (Shin et al., 2021; Marion et al., 2021) and canonical utterances (Berant and Liang, 2014; Su and Yan, 2017; Yu et al., 2020). Despite effective, these approaches either incur training inefficiency or performance loss (Cao et al., 2019). Besides, all these methods are tightly coupled to a specific dataset or logical form, thus cannot be easily adapted to other tasks or languages (Kamath and Das, 2018).

In this paper, we propose a unified intermediate representation for graph query languages, namely GraphQ IR, to resolve these issues from a novel...
Figure 1: An example of a property graph extracted from Wikidata (Vrandečić and Krötzsch, 2014). We present a relevant user query with its corresponding logical forms in different languages and GraphQ IR.

perspective. Instead of directly mapping the input utterance to the target logical form, we first translate natural language into GraphQ IR, then compile it into the target graph query language (e.g., SPARQL, KoPL, λ-DCS, Cypher, etc.). The designs of GraphQ IR weigh up the semantics of both natural and formal language by: (a) producing the IR sequences with word order rules consistent to English (Tomlin, 2014), such that the compositional semantics of IR can be more easily aligned with the natural language utterance so as to close the semantic gap; and (b) capturing the fundamental graph structures like nodes, edges and properties, such that the IR can be losslessly compiled into any downstream graph query languages so as to unify the datasets of different languages for alleviating the data bottleneck.

Therefore, language-specific grammar features that originally pose a huge obstacle to semantic parsing are now explicitly handled by the compiler, and the neural semantic parser can concentrate on the comprehension of user queries. Additionally, with our source-to-source compiler that supports bidirectional translation between GraphQ IR and various graph query languages, our work also provides a novel toolkit that can be utilized as a transpiler for unifying different datasets or as a natural language user interface for querying graph databases.

To validate the effectiveness of GraphQ IR, we conducted extensive experiments on benchmarks KQA PRO, OVERNIGHT and MetaQA. Results show that our approach can consistently outperforms the previous works by a significant margin. Especially under the compositional generalization and few-shot learning settings, our approach with GraphQ IR can demonstrate at most 11% increase on accuracy over the baselines with strong generalization abilities. Supplementary analysis further shows that GraphQ IR is easy to debug with 89% of the errors can be fixed with simple corrections.

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 to the success of semantic parsing;
- Experimental results show that our approach can achieve state-of-the-art performance across benchmarks and strong robustness even under the compositional generalization and few-shot learning settings.
- Our implemented compiler can be also utilized as a translator among different graph query languages. We will release our code and toolkit for the uses of the community.

2 GraphQ IR

We propose GraphQ IR as a novel intermediate representation that aims to bridge the semantic gap between natural and formal language as well as unify different languages to break the data bottleneck. In this section, we define the property graph, formalize the GraphQ IR, and summarize the key principles in designing GraphQ IR.

2.1 Definition

As the top of Figure 1 demonstrates, a property graph normally includes Entity (the graph nodes, e.g., Stanley Kubrick), Attribute (the node properties, e.g., date of birth), Concept (the node types, e.g., film), Relationship (the graph edges, e.g., spouse) and Qualifier (the edge properties, e.g., start time).
1. To exemplify, in Figure 1, the production of verb-object syntactic construction (Tomlin, 2014). Graph query languages into a more natural subject–
fore, we simplify the triple-based structure in match how users typically raise queries. There-
structures in correspondence to the input utterance. parser, the target IR sequence should share similar
above key structures as terminal nodes meantime
2.2.1 Diminishing structural discrepancy
close to the natural language while preserving the
Concept elements
ative clause involving the operations over graph
EntitySet
and production rules
Logical operators that perform set
relationship and attribute queries are also formulated
as relative clauses following the English expression
can be comfortably generated by a language-
model-based neural semantic parser.

Accordingly, GraphQ IR is built on top of the
above key structures as terminal nodes meantime
having its productions consistent with the composi-
tional semantics of natural language. We for-
mally define the context-free grammar of GraphQ
IR by a quadruple \( G = (V, T, S, P) \) composed of
non-terminal symbols \( V \), terminal symbols
\( T \), the start symbol \( S \) and production rules \( P \).
We specify the productions of GraphQ IR in ex-
tended Backus–Naur form (Parr, 2013) and de-
scribe a subset of GraphQ IR’s grammar in Table
1. To exemplify, in Figure 1, the production of
GraphQ IR’s EntitySet is equivalent to a rel-
ative clause involving the operations over graph
elements Concept, Relation and Entity.

2.2 Principles

We summarize several principles in designing
GraphQ IR, concluded as: presenting the semantics
close to the natural language while preserving the
structure identical to the formal language.

2.2.1 Diminishing structural discrepancy

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

To achieve this, first, the structure of IR should
match how users typically raise queries. There-
fore, 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 (\(?c \text{ name "film" treatment or not \text{ larger than } \text{ smaller than } \text{ at least } \text{ at most} \text{ Comparison operators that take value as operand} \text{ Value type constraint} \) )

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>Productions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( \rightarrow ) EntityQuery</td>
<td>Start symbol of a query sequence</td>
</tr>
<tr>
<td>EntityQuery</td>
<td>( \rightarrow ) what is EntitySet</td>
<td>Query that returns certain entities</td>
</tr>
<tr>
<td>RelationQuery</td>
<td>( \rightarrow ) what is the relation from EntitySet to EntitySet</td>
<td>Query that returns the relationship between given entities</td>
</tr>
<tr>
<td>CountQuery</td>
<td>( \rightarrow ) how many EntitySet</td>
<td>Query that returns the quantity of certain entities</td>
</tr>
<tr>
<td>EntitySet</td>
<td>( \rightarrow ) (&lt;ES&gt;) EntitySet ( \text{ LOP }) EntitySet ( &lt;/ES&gt;) ( &lt;ES&gt;) EntitySet Constraint ( &lt;/ES&gt;) ( \text{ Concept}) Entity</td>
<td>A non-empty set of entities that match certain conditions</td>
</tr>
<tr>
<td>Constraint</td>
<td>( \rightarrow ) whose Attribute COP Value that Relation DIR to EntitySet ( \text{ LOP }) EntitySet</td>
<td>Clauses that constrain the entities condition</td>
</tr>
<tr>
<td>Entity</td>
<td>( \rightarrow ) (&lt;ES&gt;) EntitySet (&lt;R&gt;) EntitySet (&lt;A&gt;) EntitySet (&lt;C&gt;) EntitySet</td>
<td>Entity node of a specified name type</td>
</tr>
<tr>
<td>Concept</td>
<td>( \rightarrow ) (&lt;ES&gt;) concept ( &lt;/ES&gt;)</td>
<td>Set of entities of a specified type</td>
</tr>
<tr>
<td>Attribute</td>
<td>( \rightarrow ) (&lt;A&gt;) attribute ( &lt;/A&gt;)</td>
<td>Specified property of an entity</td>
</tr>
<tr>
<td>Relation</td>
<td>( \rightarrow ) (&lt;R&gt;) relation ( &lt;/R&gt;)</td>
<td>Relationship between entities</td>
</tr>
<tr>
<td>Value</td>
<td>( \rightarrow ) ( \text{ VTYPE }) ( &lt;V&gt;) value ( &lt;/V&gt;) ( \text{ Attribute of Entity})</td>
<td>Specified value of certain constraints</td>
</tr>
<tr>
<td>LOP</td>
<td>( \rightarrow ) ( &lt;V&gt;) and ( &lt;/V&gt;) ( or ) ( not) ( \text{ Logical operators that perform set union, intersection or exclusion})</td>
<td>Logical operators that perform set union, intersection or exclusion</td>
</tr>
<tr>
<td>DIR</td>
<td>( \rightarrow ) ( \text{ string }) ( \text{ forward }) ( \text{ backward})</td>
<td>Relationship direction constraint</td>
</tr>
<tr>
<td>VTYPE</td>
<td>( \rightarrow ) ( \text{ number }) ( \text{ year })</td>
<td>Value type constraint</td>
</tr>
</tbody>
</table>

Table 1: A subset of GraphQ IR grammar rules. We separate alternative productions at the same level using “|” and mark the terminal nodes in italics. A more complete set of grammar is presented in Appendix Table 7.

2.2.2 Eliminating semantic redundancy and ambiguity

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

Secondly, IR should also leave out the variables
(e.g., \(?e\), \(?c\) in SPARQL) and operators (e.g.,
MATCH, DISTINCT, WHERE, RETURN, etc.) in
graph query languages that cannot be easily aligned
to natural language. Alternatively, human-readable
operators are adopted in GraphQ IR for better com-
prehension by the language models.
When designing an IR, such redundant and ambiguous semantics should be clarified into more definitive and orthogonal representation (Campagna et al., 2019). Thus in GraphQ IR’s, we unify all such unnecessary distinctions and prune redundant structure in the logical forms to distill the core semantics. In the previous example, GraphQ IR instead only requires a simple concept constraint “<C concept</C>” as the noun modifier. This not only makes the language more regular and hence simpler for both the users and semantic parser, but also facilitates the next-step compilation from IR to the downstream formal languages.

2.2.3 Inducing compilation-aiding syntax

Apart from the aforementioned designs to improve the alignment with natural language, the grammar of IR also needs to maintain the key structures in graph queries so as to enable the subsequent automatic 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 induce <ES> as a scoping token in GraphQ IR to explicitly indicate the underlying dependencies among the clauses produced by an EntitySet, as shown in Table 1. Such scoping tokens in GraphQ IR can facilitate the compiler to recover the hierarchical structure and finally transform the IR sequences into one of the graph query languages deterministically.

3 Implementation

We depict the whole picture of our proposed framework in Figure 2. The neural semantic parser first translates 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 dependencies among the input tokens. The decoder module subsequently then interprets the hidden representations and generates the IR sequence by factorizing the probability distribution:

\[
p(IR) = \prod_{i=1}^{n} p(y_i | X, y_1, ..., y_{i-1}), \quad (1)\]

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, which has a standard Transformer architecture that get pre-trained on over 160GB of English text corpus.
Therefore, BART is proficient in comprehending the diverse user utterances and generating the GraphQ IR that has natural-language-like semantic representations.

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

3.2 Compiler

The implementation of compiler comprises a front-end module that generates abstract syntax trees from GraphQ IR and a back-end module that transforms the tree structure into the target code.

Following the grammar rules formally defined in Section 2.1, the compiler front-end first performs lexical analysis and syntax analysis on the IR sequence generated in Section 3.1. The LL(*) parser that performs the leftmost derivation of a sentence with tokens look-ahead (Parr and Fisher, 2011) will automatically parse the sequence of GraphQ IR into an abstract syntax tree that contains the dependencies and hierarchical structure. The compiler back-end will then traverse the abstract syntax tree and restructure the elements and dependencies into one of the downstream graph query languages (i.e., SPARQL, KoPL, λ-DCS, Cypher, etc) following our pre-defined transformation rules. To illustrate this process, we present 2 examples of generating λ-DCS and SPARQL queries respectively in Appendix Figure 3 and Figure 4.

4 Experiments

This section evaluates the effectiveness of GraphQ IR on several benchmarks under different task settings.

4.1 Datasets

For evaluation, we choose SPARQL, KoPL, λ-DCS, and Cypher as the target graph query languages, and take KQA PRO, Overnight, MetaQA as the corresponding datasets.

KQA PRO  KQA PRO (Cao et al., 2020b) is a large-scale dataset for complex question answering over Wikidata knowledge base (Vrandečić and Krötzsch, 2014). It is so far the largest KBQA corpus that contains 117,790 natural language questions along with the corresponded SPARQL and KoPL logical forms, covering diverse question types, e.g., multi-hop inference, logical union and intersection, etc. In our experiment, it is divided into 94,376 train, 11,797 dev and 11,797 test cases.

Overnight  Overnight (Wang et al., 2015) is a semantic parsing dataset with 13,682 examples across 8 different domains extracted from Freebase (Bollacker et al., 2008). Each domain has natural language questions and pairwise λ-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/dev/test splits as in the previous work (Wang et al., 2015).

MetaQA  MetaQA (Zhang et al., 2018) is a large-scale dataset containing more than 400k multi-hop question-answer pairs over MovieQA knowledge base (Tapaswi et al., 2016). Since previous works have achieved ∼100% accuracy on its SPARQL annotation (Huang et al., 2021; Shi et al., 2021), we reconstruct METAQA into Cypher as a few-shot learning benchmark. To the best of our knowledge, this is also the first Cypher dataset in the research field of graph query language parsing.

4.2 Metric

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

4.3 Configurations

For the neural semantic parser, we used the BART\textsubscript{base} model (Lewis et al., 2020) released by Facebook on HuggingFace\textsuperscript{1}. 12 special tokens (e.g., `<ES>`) were added to the tokenizer vocabulary as the structure indicators for GraphQ IR. We used the AdamW optimizer (Loshchilov and Hutter, 2018) with the learning rate set to $3e^{-5}$ and weight decay set to $1e^{-5}$ following the default settings.

For 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\textsuperscript{2} 7.20, SEMPRE\textsuperscript{3} 2.4, Neo4j\textsuperscript{4} 4.4 and KoPL 0.3\textsuperscript{5} as the back-ends respectively for executing the SPARQL, λ-DCS, Cypher and KoPL queries.

Our whole experiments were performed on a

\textsuperscript{1}https://huggingface.co/facebook/bart-base
\textsuperscript{2}https://github.com/openlink/virtuoso-opensource
\textsuperscript{3}https://github.com/percyliang/sempre
\textsuperscript{4}https://github.com/neo4j/neo4j
\textsuperscript{5}https://pypi.org/project/KoPL/
Table 2: Experimental results 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.

<table>
<thead>
<tr>
<th>Baselines</th>
<th>Multi-hop</th>
<th>Qualifier</th>
<th>Comparison</th>
<th>Logical</th>
<th>Count</th>
<th>Verify</th>
<th>Zero-shot</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGCN (Schlichtkrull et al., 2018)</td>
<td>34.00</td>
<td>27.61</td>
<td>30.03</td>
<td>35.85</td>
<td>41.91</td>
<td>65.88</td>
<td>-</td>
<td>35.07</td>
</tr>
<tr>
<td>BART+KoPL (Cao et al., 2020b)</td>
<td>85.57</td>
<td>80.57</td>
<td>93.83</td>
<td>86.33</td>
<td>84.20</td>
<td>84.94</td>
<td>86.29</td>
<td>87.17</td>
</tr>
<tr>
<td>BART+SPARQL (Cao et al., 2020b)</td>
<td>88.49</td>
<td>83.09</td>
<td>96.12</td>
<td>88.67</td>
<td>85.78</td>
<td>92.33</td>
<td>87.88</td>
<td>89.68</td>
</tr>
<tr>
<td>CFQ IR (Herzig et al., 2021)</td>
<td>87.51</td>
<td>81.32</td>
<td>95.70</td>
<td>90.33</td>
<td>86.23</td>
<td>92.20</td>
<td>87.12</td>
<td>88.96</td>
</tr>
<tr>
<td><strong>Our Approach</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphQ IR</td>
<td>90.38</td>
<td>84.90</td>
<td>97.15</td>
<td>92.64</td>
<td>89.39</td>
<td>94.20</td>
<td>94.20</td>
<td>91.70</td>
</tr>
</tbody>
</table>

Table 3: Experimental results on OVERNIGHT dataset. Methods with asterisk (*) means that external data of other domains is utilized to enhance the performance.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SPO (Wang et al., 2015)</td>
<td>46.3</td>
<td>41.9</td>
<td>74.4</td>
<td>54.0</td>
<td>59.0</td>
<td>70.8</td>
<td>75.9</td>
<td>48.2</td>
<td>58.8</td>
</tr>
<tr>
<td>Seq2Action (Chen et al., 2018a)</td>
<td>88.2</td>
<td>61.4</td>
<td>81.5</td>
<td>74.1</td>
<td>80.7</td>
<td>82.9</td>
<td>80.7</td>
<td>82.1</td>
<td>79.0</td>
</tr>
<tr>
<td>DUAL (Cao et al., 2019)</td>
<td>84.9</td>
<td>61.2</td>
<td>78.6</td>
<td>67.2</td>
<td>78.3</td>
<td>80.6</td>
<td>78.9</td>
<td>81.3</td>
<td>76.4</td>
</tr>
<tr>
<td>2-stage DUAL* (Cao et al., 2020a)</td>
<td>87.2</td>
<td>65.7</td>
<td>80.4</td>
<td>75.7</td>
<td>80.1</td>
<td>86.1</td>
<td>82.8</td>
<td>82.7</td>
<td>80.1</td>
</tr>
<tr>
<td><strong>Our Approach</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphQ IR</td>
<td>88.2</td>
<td>64.7</td>
<td>81.6</td>
<td>72.0</td>
<td>77.6</td>
<td>83.3</td>
<td>84.9</td>
<td>81.6</td>
<td>79.5</td>
</tr>
<tr>
<td>GraphQ IR*</td>
<td>88.2</td>
<td>65.4</td>
<td>81.6</td>
<td>81.5</td>
<td>82.6</td>
<td>92.9</td>
<td>89.8</td>
<td>84.1</td>
<td>82.1</td>
</tr>
</tbody>
</table>

Table 4: Experimental results on KQA PRO compositional generalization data split.

<table>
<thead>
<tr>
<th>Baselines</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 IR</td>
<td>50.70</td>
<td>25.33</td>
<td>93.77</td>
<td>50.73</td>
</tr>
<tr>
<td>GraphQ IR</td>
<td>54.91</td>
<td>40.46</td>
<td>95.19</td>
<td>54.90</td>
</tr>
<tr>
<td>GraphQ IR*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Few-shot learning results on METAQA dataset. GraphQ IR* refers to our model that has formerly trained on KQA PRO dataset.

<table>
<thead>
<tr>
<th>Baselines</th>
<th>1-shot</th>
<th>3-shot</th>
<th>5-shot</th>
</tr>
</thead>
<tbody>
<tr>
<td>BART</td>
<td>73.93</td>
<td>91.99</td>
<td>94.37</td>
</tr>
<tr>
<td>GraphQ IR</td>
<td>72.05</td>
<td>93.73</td>
<td>95.16</td>
</tr>
<tr>
<td>GraphQ IR*</td>
<td>84.91</td>
<td>95.31</td>
<td>96.13</td>
</tr>
</tbody>
</table>

4.4 Results

As Table 2 illustrates, our proposed approach with GraphQ IR consistently outperforms the previous approaches and has achieved the new state-of-the-art results on KQA PRO dataset. In particular, GraphQ IR exhibits even larger improvements over the baselines under the more complex MULTI-HOP, QUALIFIER and ZERO-SHOT task settings. We attribute this to its natural-language-like semantic representations that can be well accepted by the pretrained language model and its formal-language-like syntactic structure that can be flexibly combined or decomposed to achieve better generalization.

As for the benchmark OVERNIGHT, our approaches with GraphQ IR also significantly surpass the baselines, as shown in Table 3. Due to the distinct vocabulary and grammar rules, previous works usually train separate parsers for each task domain (Wang et al., 2015; Chen et al., 2018a). However, as can be observed from the superior performance of GraphQ IR* that get trained on the aggregated data of all eight domains, with an extra layer of IR for unification, domain-specific grammar rules are now consolidated into one universal representation and the training of one domain can thereby benefit from the others’ data.

To further validate our proposed IR’s robustness, we also examine GraphQ IR under compositional generalization and few-shot learning task settings.

Compositional Generalization Current neural semantic parsers often fail in reaching good compositional generalization, i.e., the capability of generalizing from the known components to produce novel combinations (Pasupat and Liang, 2015; Keysers et al., 2019; Furrer et al., 2020). To measure our IR’s compositional generalization ability, we create a new KQA PRO data split based on the logical form length and test the parsers to generate long
The results in Table 5 indicate that our IR can generalize well with very few labelled data. Moreover, the GraphIR® model that has in advance trained on KQA PRO data demonstrates the most outstanding performance, especially under the hardest 1-shot setting. This further affirms the necessities of unification. With GraphIR, the annotations originally prepared for other graph query languages (e.g., SPARQL) can be now easily transferred to facilitate a novel task domain lack of data.

## 5 Error Analysis

To investigate GraphIR’s potentials and bottleneck, 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:

<table>
<thead>
<tr>
<th>Error Type</th>
<th># Errors</th>
<th># OSCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inaccurate data annotation</td>
<td>28</td>
<td>27</td>
</tr>
<tr>
<td>Ambiguous query expression</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>Unspecified graph structure</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>Nonequivalent semantics</td>
<td>32</td>
<td>25</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.

queries based on the short query components seen in the training data. The results are listed in Table 4. Comparing with the plain-BART baseline and the CFQ IR (Herzig et al., 2021) that is specially designed for improving the compositional generalization on SPARQL, GraphIR achieves the best performance in overall data as well as in complex task settings, which can be again credited to our IR.

### Few-shot learning

In practice, it is important for a parser to remain robust in a novel task domain lack of data annotations. Therefore, we reconstruct the METAQA dataset into Cypher, a graph query language commonly used in the industries but rarely studied in previous semantic parsing research works (Seifer et al., 2019), and assess our models under the few-shot learning setting. We adjust the data to ensure only 1, 3 and 5 samples of each question type appear in the training set respectively under the 1-shot, 3-shot and 5-shot settings. The results in Table 5 indicate that our IR can generalize well with very few labelled data. Moreover, the GraphIR® model that has in advance trained on KQA PRO data demonstrates the most outstanding performance, especially under the hardest 1-shot setting. This further affirms the necessities of unification. With GraphIR, the annotations originally prepared for other graph query languages (e.g., SPARQL) can be now easily transferred to facilitate a novel task domain lack of data.

### Few-shot learning

In practice, it is important for a parser to remain robust in a novel task domain lack of data annotations. Therefore, we reconstruct the METAQA dataset into Cypher, a graph query language commonly used in the industries but rarely studied in previous semantic parsing research works (Seifer et al., 2019), and assess our models under the few-shot learning setting. We adjust the data to ensure only 1, 3 and 5 samples of each question type appear in the training set respectively under the 1-shot, 3-shot and 5-shot settings. The results in Table 5 indicate that our IR can generalize well with very few labelled data. Moreover, the GraphIR® model that has in advance trained on KQA PRO data demonstrates the most outstanding performance, especially under the hardest 1-shot setting. This further affirms the necessities of unification. With GraphIR, the annotations originally prepared for other graph query languages (e.g., SPARQL) can be now easily transferred to facilitate a novel task domain lack of data.

### Few-shot learning

In practice, it is important for a parser to remain robust in a novel task domain lack of data annotations. Therefore, we reconstruct the METAQA dataset into Cypher, a graph query language commonly used in the industries but rarely studied in previous semantic parsing research works (Seifer et al., 2019), and assess our models under the few-shot learning setting. We adjust the data to ensure only 1, 3 and 5 samples of each question type appear in the training set respectively under the 1-shot, 3-shot and 5-shot settings. The results in Table 5 indicate that our IR can generalize well with very few labelled data. Moreover, the GraphIR® model that has in advance trained on KQA PRO data demonstrates the most outstanding performance, especially under the hardest 1-shot setting. This further affirms the necessities of unification. With GraphIR, the annotations originally prepared for other graph query languages (e.g., SPARQL) can be now easily transferred to facilitate a novel task domain lack of data.

### Few-shot learning

In practice, it is important for a parser to remain robust in a novel task domain lack of data annotations. Therefore, we reconstruct the METAQA dataset into Cypher, a graph query language commonly used in the industries but rarely studied in previous semantic parsing research works (Seifer et al., 2019), and assess our models under the few-shot learning setting. We adjust the data to ensure only 1, 3 and 5 samples of each question type appear in the training set respectively under the 1-shot, 3-shot and 5-shot settings. The results in Table 5 indicate that our IR can generalize well with very few labelled data. Moreover, the GraphIR® model that has in advance trained on KQA PRO data demonstrates the most outstanding performance, especially under the hardest 1-shot setting. This further affirms the necessities of unification. With GraphIR, the annotations originally prepared for other graph query languages (e.g., SPARQL) can be now easily transferred to facilitate a novel task domain lack of data.

### Few-shot learning

In practice, it is important for a parser to remain robust in a novel task domain lack of data annotations. Therefore, we reconstruct the METAQA dataset into Cypher, a graph query language commonly used in the industries but rarely studied in previous semantic parsing research works (Seifer et al., 2019), and assess our models under the few-shot learning setting. We adjust the data to ensure only 1, 3 and 5 samples of each question type appear in the training set respectively under the 1-shot, 3-shot and 5-shot settings. The results in Table 5 indicate that our IR can generalize well with very few labelled data. Moreover, the GraphIR® model that has in advance trained on KQA PRO data demonstrates the most outstanding performance, especially under the hardest 1-shot setting. This further affirms the necessities of unification. With GraphIR, the annotations originally prepared for other graph query languages (e.g., SPARQL) can be now easily transferred to facilitate a novel task domain lack of data.
in a directed cycle graph, but some of them contain structures that are absent in a knowledge base. (4) **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 data annotation or one step correction on the GraphQ IR element, demonstrating that our proposed approach with GraphQ IR can generate high quality logical forms and is 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), λ-DCS (Liang, 2013; Pasupat and Liang, 2015; Wang et al., 2015), SQL (Zhong et al., 2017; Yu et al., 2020), AMR (Banarescu et al., 2013), SPARQL (Sun et al., 2020) and KoPL (Cao et al., 2020b, 2021). Most recent works take semantic parsing as a Seq2Seq translation task via 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., 2020b); 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 semantic parser adapting 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., 2007). 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 in-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), whereas another category of works instead construct 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 same elements (2021).

Although these works partially mitigate the mismatch between natural and formal language, they either neglected the structural information requisite for downstream compilation or failed in removing the formal representations that are unnatural to the language models. In contrast, GraphQ IR can benefit from both approaches with its natural-language-like semantic representation and formal-language-like syntactic definition. The unification also saves the huge costs as incurred when preparing separate IR and compiler for each graph query languages in previous works.

7 Conclusion and Future Work

This paper proposes a novel intermediate representation, namely GraphQ IR, for bridging the structural and semantic gap between natural language and graph query languages. Evaluation results show that our approach using GraphQ IR consistently surpasses the baselines on multiple benchmarks covering different formal languages, i.e., SPARQL, KoPL, λ-DCS and Cypher. Moreover, GraphQ IR also demonstrates superior generalization ability and robustness under the compositional generalization and few-shot learning settings.

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


Shulin Cao, Jiaxin Shi, Zijun Yao, Xin Lv, Jifan Yu, Lei Hou, Juan-Zi Li, Zhiyuan Liu, and Jinghui Xiao. 2021. Program transfer for answering complex questions over knowledge bases.


Table 7: A more complete set of GraphQIR grammar rules that covers the common graph query patterns. Italic words refer to the terminal symbols. For simplicity, here we omit the special production rules for handling the corner cases.
User 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 3: A user query in OVERNIGHT. As aforementioned in Section 3, the neural semantic parser first translates the input utterance into the GraphQ IR. The front-end of compiler then parses the GraphQ IR sequence into an abstract syntax tree, which is subsequently transformed into the corresponding $\lambda$-DCS sequence by the compiler back-end.
User 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 <ES> <ES> ones that <R> filming location </R> forward to <E> To Rome with Love </ES> <ES> <ES> or <ES> <E> Lisbon </E> <ES> ones that <R> twinned administrative body </R> forward to <E> Santo Domingo </E> </ES> </ES> </ES>

GraphQ IR abstract syntax tree:

SPARQL abstract syntax tree:

SPARQL sequence:

Figure 4: A user query in KQA PRO. Similarly, the compiler parses the generated GraphQ IR sequence into an abstract syntax tree, then transform its tree structure into the corresponding SPARQL sequence.