What Is the Cube Root of 27? Question Answering Over CodeOntology

Mattia Atzeni and Maurizio Atzori(✉)

Math/CS Department, University of Cagliari,
Via Ospedale 72, 09124 Cagliari, CA, Italy
m.atzeni38@studenti.unica.it, atzori@unica.it

Abstract. We present an unsupervised approach to process natural language questions that cannot be answered by factual question answering nor advanced data querying, requiring instead ad-hoc code generation and execution. To address this challenging task, our system, AskCO, performs language-to-code translation by interpreting the natural language question and generating a SPARQL query that is run against *CodeOntology*, a large RDF repository containing millions of triples representing Java code constructs. The query retrieves a number of Java source code snippets and methods, ranked by AskCO on both syntactic and semantic features, to find the best candidate, that is then executed to get the correct answer. The evaluation of the system is based on a dataset extracted from StackOverflow and experimental results show that our approach is comparable with other state-of-the-art proprietary systems, such as the closed-source WolframAlpha computational knowledge engine.

Keywords: Question answering over linked data
Natural language programming · Semantic parsing · Machine reading
Language-to-code

1 Introduction

Question Answering over Linked Data and ontologies allows leveraging structured data and Natural Language Processing to give a precise answer to the input provided by the end user. However, most of the information available in the Web is organized in the form of unstructured or semi-structured data, thereby being difficult to be automatically processed by such approaches. A paradigmatic example is represented by massive open source code repositories, where source code is not readily available to be queried as Linked Open Data, despite the great potential for the development of computational knowledge engines capable of leveraging this impressive amount of information. To overcome this issue, we have recently introduced CodeOntology[1] [1,2], as a resource aimed at allowing the adoption of the Semantic Web technology stack within the domain of

[1] http://codeontology.org.

software development and engineering. CodeOntology consists of two main contributions *(i)* an ontology modeling object-oriented code constructs and *(ii)* a parser which is capable of analyzing Java source code and serializing it into RDF triples. CodeOntology also includes a dataset containing millions of RDF triples extracted from OpenJDK [3].

Following this research line, in this paper we introduce an algorithmic approach that addresses the task of Natural Language Programming by employing CodeOntology for Question Answering over source code. Hence, we target a Question Answering problem where the answer to the input question is not directly available in the data, but the dataset contains the information that is needed to *compute* the correct answer. This challenging task is accomplished by performing an unsupervised semantic parsing of natural language utterances into a Java source code, which can be automatically executed to retrieve the answer to the input question.

We discuss two approaches: *(i)* a fast coarse-grained approach which only supports natural language commands corresponding to the invocation of a single method, and *(ii)* a fine-grained approach which is based on dependency parsing and is capable of tagging substrings of the input question with entities from CodeOntology, thereby supporting the execution of more complex expressions, involving the invocation of multiple methods. Within the coarse-grained approach, we propose a simple technique to rank entities available in CodeOntology (specifically, Java methods), based on syntactic and semantic features. On the one hand, the first approach is aimed at providing a natural language interface to Java source code, focusing on applications for developers, such as Computer Assisted Coding tools pluggable within IDEs. Hence, we assume that the user can specify a description of the method to be invoked and the actual arguments. These arguments can be of any arbitrary type, including user-defined classes. On the other hand, the fine-grained approach is aimed at providing a computational knowledge engine for Question Answering and other end-user applications, such as speech-driven tools like Amazon Alexa. Hence, we assume the input is a single natural language question and the actual arguments are provided within the question as literals, thereby limiting the type of the parameters that the user can effectively specify.

Experimental results are based on a dataset extracted from StackOverflow and show that our approach is comparable with state-of-the-art systems, such as the proprietary closed-source WolframAlpha computational knowledge engine. Thus, the main contributions of this work are:

– we introduce an unsupervised approach capable of mapping natural language utterances into Java source code, by leveraging the possibility of extracting Linked Data from any Java project;
– we propose a technique to rank entities from CodeOntology (Java methods) based on syntactic and semantic features;
– we provide a dataset derived from simple questions extracted from StackOverflow, to evaluate the performances of our system[2].

[2] available at: https://doi.org/10.6084/m9.figshare.6071663.

We remark that, while the paper focuses on OpenJDK methods only, the resulting system, that we called AskCO, is general enough to be applied with any custom set of Java repositories.

2 Related Work

Natural language represents certainly one of the easiest ways to interact with a computer for humans. In Question Answering over Linked Data (QALD), indeed, natural language questions are translated into SPARQL queries to find factual information or more advanced statistics from, e.g., datacubes [4]. Although falling in the area of QALD, our work focuses on questions for which no answer can be found by only querying a repository, since the correct answer needs to be computed by generating and executing code.

In this sense, this work resembles more closely related approaches to natural language querying in software engineering. A large body of work has been done to allow software engineers to manage information about large software systems. For instance, LaSSIE [5] was a prototype tool which made use of a frame-based description language, as well as explicit knowledge representation and reasoning, to address the problem of discovering and learning new information about an existing system. LaSSIE was also embedded with a simple natural language interface based on a taxonomy of the domain and on a lexicon, which included the words known to the system. This work has inspired several more recent research projects, such as [6], where Semantic Web technologies have been applied to support *guided-input natural language* queries concerning static source code information. The presented approach allows importing knowledge about the evolution of a software system into a RHDB (Release History Database), which is augmented with ontological information on source code. Although similar to our work, the expressiveness of this approach is in fact limited by the kind of questions it supports, as it relies on Ginseng [7] to constrain the input and answer quasi-natural language queries by leveraging a multi-level grammar which defines the structure of supported sentences. Similarly, in [8] an unambiguous and controlled subset of natural language with a restricted grammar and a domain-specific vocabulary is used to run queries for static information on source code. On the other hand, more advanced approaches have been developed to support unconstrained natural language queries. In [9], indeed, natural language processing (NLP) techniques are applied to translate free questions to concrete parameters of a third-party query engine.

All the approaches outlined so far are mainly aimed at retrieving static information like specific method calls or write access to certain fields. Our technical contribution describes instead a novel algorithm which brings together NLP and Semantic Web technologies to translate natural language into object-oriented source code. Several research prototypes have been developed to enable the automatic understanding of a natural language description of a program. For instance, Metafor [10], based on concepts from *Programmatic Semantics* [11], is capable of generating class descriptions with attributes and methods. However, its expressiveness is still limited, in the sense that it does not feature the possibility of processing arbitrary English statements. Instead, it can parse a reasonably expressive subset of the English language, to create scaffolding code fragments that can be used to assist the development process. In this sense, it is deeply different from our approach, which aims at mapping any natural language question into the execution of methods extracted from CodeOntology.

More recently, in 2017, SemEval hosted an ambitious challenge [12], aiming at supporting the interaction between users and software APIs, micro-services and applications, using natural language. Most of the work in this area has focused on supervised approaches [13], thereby requiring a dataset mapping natural language to a formal meaning representation. However, this task is different from any previous work related to semantic parsing of natural language commands, as it involves generic programming scenarios and a more comprehensive knowledge base of possible actions. A related problem was also addressed in [14], which targeted the creation of an *if-this-than-that* recipe on the IFTTT[3] platform. The task outlined within the SemEval competition, however, is even more challenging, as it is not limited to *if-then* rules, and it also involves instantiating parameter values. Nevertheless, both approaches are placed in a simplified landscape with respect to our system, which aims at mapping natural language utterances into a real-world and Turing-equivalent programming language.

3 Coarse-Grained Approach

This section describes the coarse-grained approach, which is meant to allow the execution of Java methods, given a natural language description of the intended behavior. The output of such approach is a ranking of the methods in the dataset, based on a metric involving both syntactic and semantic measures. This approach is preliminary to the fine-grained one, which is instead designed to answer more complex questions.

3.1 A Natural Language Interface to OpenJDK

Although CodeOntology already features the possibility of querying source code in a semantic framework powered by the Web of Data, this capability is in fact limited by the complexity of SPARQL queries. Hence, the coarse-grained approach is aimed at providing an easy-to-use and intuitive natural language interface to the entities made available by CodeOntology. We target a RDF repository extracted from OpenJDK 8 [3], containing millions of RDF triples about structural information on source code, actual source code as literals, comments, and semantic links to DBpedia [15] resources.

In particular, we want to allow the end-user to remotely search and execute methods available in the dataset, without necessarily knowing the signature of the method, but only its intended behavior. Thus, we assume that the end-user can provide: *(i)* a natural language description of the method; *(ii)* an unsorted

[3] https://ifttt.com/.

list of the actual parameters, optionally including, if the method is not static, the target instance of the method invocation; *(iii)* the expected return type. The system should then run a SPARQL query on the RDF dataset, searching for methods from OpenJDK, whose signature is compatible with the values specified by the user. The retrieved results are subsequently ranked to select the method which most closely matches the natural language description. The selected method is then invoked on the specified input parameters, and the result is then returned to the user, along with the ranking produced by the system. Figure 1 shows the result of the application of the ranking process within the coarse-grained approach.

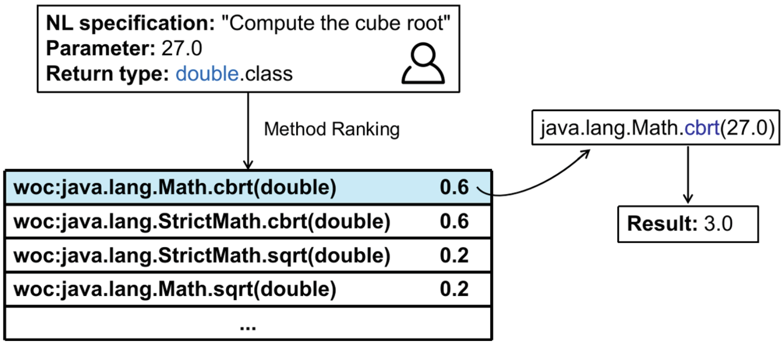
NL specification: "Compute the cube root"
Parameter: 27.0
Return type: double.class
Method Ranking
woc:java.lang.Math.cbrt(double) 0.6
woc:java.lang.StrictMath.cbrt(double) 0.6
woc:java.lang.StrictMath.sqrt(double) 0.2
woc:java.lang.Math.sqrt(double) 0.2
...
java.lang.Math.cbrt(27.0)
Result: 3.0

Fig. 1. Example of a simple application of the coarse-grained approach.

3.2 Method Ranking

The ranking of the methods in the dataset relies on the following attributes: *(i)* the name of the method; *(ii)* the Javadoc comment associated with the method; *(iii)* the name of the declaring class; *(iv)* semantic links to DBpedia, already provided by CodeOntology. Several similarity measures are used to produce the final ranking. Such measures are used both at syntactic and semantic level.

Syntactic measures are based on the name of the method, the name of the declaring class and code comments. In particular, the natural language description of the behavior of the method is pre-processed using a standard NLP pipeline which performs sentence splitting, tokenization and lemmatization. Next, we compute the following measures:

- **LS:** normalized Levenshtein similarity against the name of the method;
- **COM:** n-gram overlap against the Javadoc comment related to the method;
- **CN:** n-gram overlap against the name of the declaring class.

More precisely, given two sets S_1 and S_2 of consecutive n-grams from two different sentences, the n-gram overlap is defined as:

$$ngo(S_1, S_2) = 2 \cdot \left(\frac{|S_1|}{|S_1 \cap S_2|} + \frac{|S_2|}{|S_1 \cap S_2|} \right)^{-1}.$$

Thus, the n-gram overlap is computed as the harmonic mean of the degree to which the second sentence covers the first and the degree to which the first sentence covers the second. In practice, for n-grams we set $n = 1$. On the other hand, the Levenshtein distance d_L between two strings is defined as the minimum number of single-character edits, required to change one string into the other. Since we need a similarity value in the range between 0 and 1, we compute the normalized Levenshtein similarity as:

$$s_L(s_1, s_2) = 1 - \frac{d_L(s_1, s_2)}{\max\{|s_1|, |s_2|\}}.$$

Levenshtein distance and n-gram overlap are used to match methods from OpenJDK and the natural language command provided by the user at a syntactic level. To incorporate semantics into the ranking process, we leverage DBpedia links readily available in the dataset and word embeddings to comute the following features:

- **NED:** ratio of the DBpedia links shared by the comment of the method and the natural language command;
- **W2V:** cosine similarity between the mean vector associated with the comment of the method and the mean vector associated with the natural language command.

More precisely, we make use of TagMe [16], to perform Named Entity Disambiguation on the input text and retrieve a set of links to DBpedia resources. Each method available in the dataset provides DBpedia links generated using the same approach, applied on the Javadoc comment. Hence, we use the ratio of the shared links as a measure of semantic relatedness between each retrieved method and the input command.

Moreover, we apply a Word2Vec [17] pre-trained model to retrieve 300-dimensional word vectors from each word in both the natural language specification provided by the user and the comment associated with methods in CodeOntology. The cosine similarity between the mean vector corresponding to the input command and the mean vector associated with each Java method is used as another semantic measure. The final score applied within the ranking process is the average value of the syntactic and semantic measures described so far.

4 Fine-Grained Approach

As we have already mentioned, the fine-grained approach is aimed at dealing with more complex natural language utterances, possibly involving the execution of more methods. Given a question in natural language, this approach is capable of parsing the input question into a Java source code which gets executed to produce the desired answer. This section details how this approach actually works and how it can be used for question answering over source code.

4.1 Dependency Graph Unfolding

Given a natural language question, the fine-grained approach starts by applying Stanford CoreNLP [18] to perform dependency parsing. We assume that the question provided by the user may include primitive literals, such as string literals, integers, Booleans, and parameters of type double. Hence, before parsing the input sentence, care must be taken to replace string literals with a placeholder, in order to prevent the dependency parser from processing also actual arguments. The output of such process is the graph of the dependencies, as shown in Fig. 2.

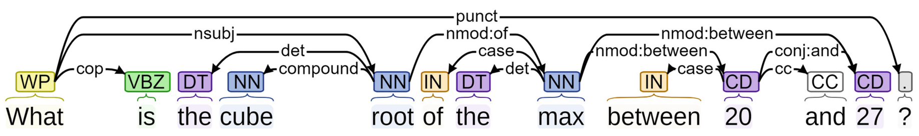

Fig. 2. Result of dependency parsing on a simple input question.

This graph is unfolded into a tree and pruned to remove nodes that are not useful for our purposes. In particular, we allow merging two nodes, depending on the nature of the dependency between the corresponding words. For instance, multiword expressions (MWEs) are merged into a single node, and, similarly, adjectival or adverbial modifiers are joined with the word they refer to. We also allow removing leaf nodes such as conjunctions, determiners and punctuation.

The result is further post-processed, to ensure that all the literal arguments specified by the user correspond to some leaf node of the tree and that no subtree is repeated. Figure 3 shows the result of the application of this approach to the graph depicted in Fig. 2.

4.2 Mapping to a Feasible Execution Tree

The unfolding of the dependency graph results in a tree, such that the set of nodes N can be partitioned into two subsets $\mathcal{L}$ and $\mathcal{M}$, where *(i)* $\mathcal{L}$ is the subset of nodes corresponding to literal actual arguments, *(ii)* $\mathcal{M}$ is the subset of nodes corresponding to natural language utterances denoting a method invocation, *(iii)* each node in $\mathcal{L}$ is a leaf, *(iv)* $N = \mathcal{L} \cup \mathcal{M}$ and $\mathcal{L} \cap \mathcal{M} = \emptyset$.

We want to obtain a tree where each node $i \in \mathcal{M}$ is labeled with a method ranking $\mathcal{R}_i$, that is a sequence $(m_1, s_1) \cdots (m_n, s_n)$, such that *(i)* m_i is a Method for all $i = 1 \ldots n$, *(ii)* $s_i \in [0, 1]$ for all $i = 1 \ldots n$, *(iii)* $i < j \Rightarrow s_i \geq s_j$. To do this, we need to query CodeOntology and rank methods using the coarse-grained approach. However, we only have to select methods whose signature is compatible with the structure of the tree and with the actual arguments provided by the user. Hence, we also label each node with the set of the types it can assume. To this end, we define the set $\mathsf{Types} = \{t \in \mathcal{K} \mid t : T \wedge T \sqsubseteq \mathsf{Type}\}$, as the set of all types available in our knowledge base $\mathcal{K}$. Next, we define the function *types* $: N \to 2^{\mathsf{Types}}$, such that:

$$types(i) = \begin{cases} t & \text{if } i \in \mathcal{L} \text{ and } t \text{ is the type of } i \\ returnTypes(\mathcal{R}_i) & \text{if } i \in \mathcal{M} \end{cases},$$

where $returnTypes(\mathcal{R}_i)$ can be computed as:

$$returnTypes(\mathcal{R}) = \begin{cases} \{r\} \cup returnTypes(\mathcal{R}') & \text{if } \mathcal{R} = \mathcal{R}'(m, s) \text{ and} \\ & m \text{ has return type } r \\ \emptyset & \text{if } \mathcal{R} = [] \end{cases}.$$

To assign these type labels to the nodes, we start from the leaves, as each node in $\mathcal{L}$ can be labeled with the CodeOntology resource associated with its type. Next, we can recursively label with a set of types also each node in $\mathcal{M}$, by employing the following approach. We select the nodes such that their children have already been labeled with a set of types and we query CodeOntology for methods that are compatible with the specified arguments. The list of arguments may be unsorted and may also include the target instance of the method invocation. The retrieved methods are then ranked as described in Sect. 3.2 and the corresponding node is labeled with the set of their return types. Algorithm 1 details the described approach.

Algorithm 1. *RankOnTree*(i)

```
1  if $i \in \mathcal{L}$ then
2      Let $t$ be the type of $i$
3      $types(i) \leftarrow \{t\}$
4  else
5      Let $\boldsymbol{l} = [l_1, \ldots, l_n]$ be the list of the children of $i$
6      foreach $l_j \in \boldsymbol{l}$ do
7          $RankOnTree(l_j)$
8      end
9      Let $\mathbf{t} = [t_1, ..., t_n]$ be a list such that $t_j = types(l_j)$ for each $l_j \in \boldsymbol{l}$
10     Query CodeOntology for methods whose signature is compatible with $\boldsymbol{t}$
11     Let $\mathcal{R}_i$ be the ranking of the resulting methods, computed using the
       coarse-grained approach
12     $types(i) \leftarrow returnTypes(\mathcal{R}_i)$
13 end
```

After applying Algorithm 1 on the result of the dependency graph unfolding, we obtain a new tree structure, where each node in $\mathcal{M}$ is labeled with a ranking of methods retrieved from CodeOntology. Figure 3 shows an example of such a tree.

Now, we want to select a method from each ranking, in such a way that the combination of all the selected methods is feasible, meaning that it corresponds to compilable Java source code. At the same time, however, we also need to

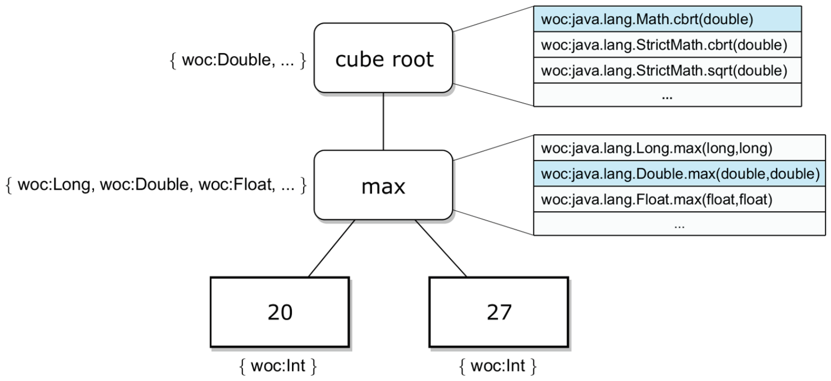
{ woc:Double, ... }
cube root
woc:java.lang.Math.cbrt(double)
woc:java.lang.StrictMath.cbrt(double)
woc:java.lang.StrictMath.sqrt(double)
...
{ woc:Long, woc:Double, woc:Float, ... }
max
woc:java.lang.Long.max(long,long)
woc:java.lang.Double.max(double,double)
woc:java.lang.Float.max(float,float)
...
20
{ woc:Int }
27
{ woc:Int }

Fig. 3. Mapping to a feasible execution tree.

maximize the total score associated with selected methods. Hence, we have to solve the following integer linear programming problem, where $x_{ij} = 1$ if the j-th method in the ranking $\mathcal{R}_i$ is selected, and $x_{ij} = 0$ otherwise:

$$\text{Maximize} \sum_{i \in \mathcal{M}} \sum_{(m_{ij}, s_{ij}) \in \mathcal{R}_i} x_{ij} \cdot s_{ij} \tag{1}$$

subject to the following constraints: *(i)* $x_{ij} \in \{0, 1\}$, *(ii)* $\sum_j x_{ij} = 1$, for all $i \in \mathcal{M}, 1 \leq j \leq |\mathcal{R}_i|$ and *(iii)* the combination of the selected methods can be compiled.

If a solution to this problem exists, then we can turn the tree into Java source code, which gets executed to answer the original question. Moreover, the average score of selected methods can be interpreted as a measure of the confidence level about the correctness of the solution. The result of such approach is shown in Fig. 3, where selected methods have been highlighted.

4.3 Greedy Search

The algorithmic approach described up to this point may fail to return a correct answer, whenever the tree produced by unfolding the dependency graph cannot be matched to the Java source code corresponding to the input question. In particular, we want to improve the algorithm, so that it is robust to two kinds of situations: *(i)* the tree resulting from the process described in Sect. 4.1 is too detailed, meaning that it has more nodes corresponding to method invocations than needed, or *(ii)* the dependency graph produced by Stanford CoreNLP contains some errors, which can be detected by leveraging knowledge about methods in CodeOntology and typing. There are several ways to extract a tree from the input sentence and, for each tree, several combinations of methods need to be explored. This creates an intractable search space for possible solutions, and, subsequently, we cannot afford an exhaustive search. Thus, we apply a heuristic approach that, starting from the output of the process described in Sect. 4.2, performs a greedy search for better solutions. We define the following move operators that are used to turn a tree into a different configuration:

– **Merge:** two adjacent nodes in $\mathcal{M}$ are merged, the natural language utterances corresponding to such nodes are joined and the children of the newly created node are the union of the children of the merged nodes;
– **Push:** a node in $\mathcal{M}$ is pushed down or up a level in the tree, along with all its children;
– **MoveLiterals:** the children in $\mathcal{L}$ of a node in $\mathcal{M}$ are moved to a different node in $\mathcal{M}$.

Intuitively, the first move allows the algorithm to deal with trees where a single method invocation is spread across multiple nodes, while the other operators are used to handle errors in dependency parsing.

We define the distance between two trees $\mathcal{T}$ and $\mathcal{T}'$, denoted as $TED(\mathcal{T}, \mathcal{T}')$, as the minimum number of moves required to turn one tree into the other. Next, we denote the normalized distance as:

$$NTED(\mathcal{T}, \mathcal{T}') = \frac{TED(\mathcal{T}, \mathcal{T}')}{\max\{|\mathcal{T}|, |\mathcal{T}'|\}},$$

where $|\mathcal{T}|$ is the total number of nodes in $\mathcal{T}$.

Starting from an initial tree $\mathcal{T}_0$, produced as described in Sect. 4.2, the algorithm evaluates all the possible defined moves and applies a greedy search with a *Best-Improvement* strategy, in order to maximize, under the same constraints defined for Eq. 1, the following objective function:

$$z(\mathcal{T}_k) = \frac{1}{|\mathcal{M}_k|} \cdot \sum_{i \in \mathcal{M}_k} \sum_{(m_{ij}, s_{ij}) \in \mathcal{R}_i^k} x_{ij} \cdot s_{ij} - \lambda \cdot NTED(\mathcal{T}_k, \mathcal{T}_0), \qquad (2)$$

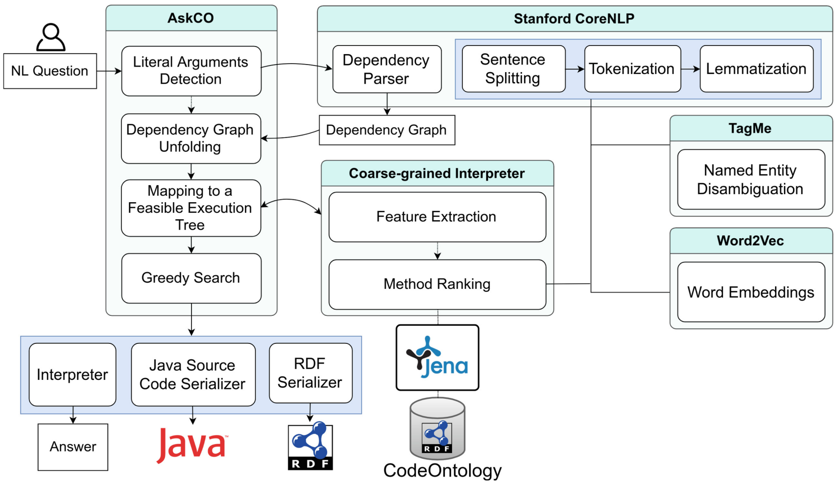

Fig. 4. High-level view of the architecture of the system.

where $\lambda \in [0, 1]$ is a constant, $\mathcal{M}_k$ is the set of non-literal nodes in $\mathcal{T}_k$ and $\mathcal{R}_i^k$ is the method ranking associated with node i in $\mathcal{T}_k$. Overall, Eq. 2 is structured as Eq. 1, with a penalization term which decreases the objective value for trees that are too different from the original tree $\mathcal{T}_0$. In practice, we set λ to 0.5. The algorithm stops when a local optimum is reached and no move can be applied to improve the objective value. Figure 4 shows a high-level view of the architecture of the system, which is available on GitHub at https://github.com/codeontology/question-answering.

5 Experiments

This section provides an evaluation for both the coarse-grained and the fine-grained approaches. Experimental results show that both techniques can be effectively applied on a RDF dataset extracted from OpenJDK 8 [3], with promising results.

5.1 Method Ranking Evaluation

The system implemented for the coarse-grained approach aims at retrieving and ranking Java methods defined within the OpenJDK 8 source code, given a natural language description of the behavior of the method. Providing an evaluation for this coarse-grained ranking of Java methods is challenging, because we are not aware of any dataset pairing natural language commands, with a corresponding set of relevant methods from OpenJDK. Hence, we have extracted a benchmark dataset containing simple questions discussed on StackOverflow[4].

The dataset has been generated by retrieving the most popular questions about the Java programming language, which have been manually filtered to select only the top 122 questions that can be answered with the invocation of a single method from OpenJDK.

For some questions, we may have more than one relevant method, so the dataset has been further manually enriched with missing methods. For instance, the natural language command *"convert a string to an integer"* is associated to two methods, namely the method `java.lang.Integer.parseInt(java.lang.String)` and the method `java.lang.Integer.valueOf(java.lang.String)`.

Overall, for more than 80% of the questions there is only one relevant method, while some question has even 3 or 4 relevant methods. The dataset is available on figshare[5] under Creative Commons Attribution 4.0 license.

We experiment several combinations of the syntactic and semantic features defined in Sect. 3.2. Table 1 reports the experimental results obtained for the coarse-grained approach. We evaluate the performance of the system based on the Mean Average Precision (MAP) obtained by the produced rankings. However, it is crucial that the first method in the ranking is correct, as it is invoked

[4] https://stackoverflow.com/.

[5] https://doi.org/10.6084/m9.figshare.6071663.

by the coarse-grained system. Thus, we also compute the precision at 1 for each ranking, and we report the mean result in Table 1 (MAP@1).

Table 1. Experimental results on method ranking

	Features	MAP@1	MAP
Syntactic features	LS	0.697	0.776
	LS + CN	0.713	0.785
	LS + COM	0.861	0.891
	LS + CN + COM	0.869	0.897
Semantic features	NED	0.607	0.714
	W2V	0.738	0.818
	W2V + NED	0.754	0.822
Syntactic + Semantic features	LS + W2V	0.795	0.852
	LS + W2V + NED	0.803	0.861
	LS + CN + COM + W2V	0.902	0.921
	LS + CN + COM + W2V + NED	**0.902**	**0.923**

As we can see, the best results are obtained by boosting syntactic features with semantics. The coarse-grained approach to the ranking of Java methods, in this case, achieves a Mean Average Precision of 0.923. At the same time, the system is capable of finding and invoking the correct method for the majority of the natural language commands available in the dataset, obtaining a MAP@1 of 0.902.

5.2 Question Answering Evaluation

Experiments on the ranking of Java methods provide a partial evaluation also for the fine-grained approach, as method ranking is the most important step for parsing natural language questions involving the invocation of multiple methods. However, to provide a further evaluation of our fine-grained system, we perform experiments on another benchmark dataset[6] we created, containing 120 questions on mathematical expressions and string manipulation. We can classify each question in the dataset by the number of methods required to provide the correct answer. We obtain that the dataset contains:

- 16 questions requiring the invocation of 1 method;
- 63 questions requiring the invocation of 2 methods;
- 36 questions requiring the invocation of 3 methods;
- 5 questions requiring the invocation of 4 methods.

Hence, the majority of the questions involves the invocation of 2 methods and, on average, 2.25 methods per question are required.

[6] available online at: https://doi.org/10.6084/m9.figshare.6071729.

We apply a threshold $t \in [0, 1]$ on the objective value defined by Eq. 2, in order to detect questions that our system is not able to process correctly. When $t = 0$, then the system will provide an answer to all questions in the dataset, while $t = 1$ means that the system basically refuses to process any question. Figure 5 summarizes the performances of the system in response to changes in the value of the threshold. As we can see, when $t = 0$ the system is capable of answering correctly 91% of the questions in the dataset. However, we can increase precision over processed questions using a higher threshold. In particular, setting a threshold $t = 0.15$ allows to get a precision over processed questions of 0.94, while leaving the global result unchanged. When precision over processed questions eventually reaches 1, then global precision equals the rate of processed questions, as clearly shown in Fig. 5.

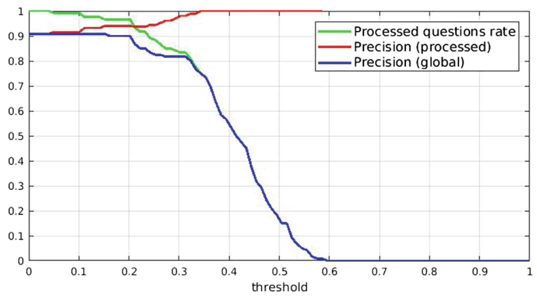

Fig. 5. Performances of the system for different values of the threshold.

It is also interesting to discuss the average size of the rankings, which contain all methods from OpenJDK whose signature is compatible with the actual arguments specified in the natural language question. At this remark, we notice that, on average, the rankings of methods produced by the fine-grained approach on this dataset contain 246.5 methods. The longest ranking includes 677 methods, while the shortest one has 24 methods. Hence, the distribution has a high standard deviation equal to 176.7 methods.

We can compare our approach with the results obtained by the WolframAlpha computational knowledge engine[7]. Of course, our system and WolframAlpha have different capabilities. On the one hand, WolframAlpha can answer a wide range of complex open-domain questions, which cannot be answered by simply invoking methods from OpenJDK. On the other hand, our system is capable of executing natural language commands which are certainly out of the scope of WolframAlpha. However, both approaches should be able to process and answer questions involving mathematical expressions and string manipulation. Table 2 shows the experimental results of the comparison between the systems.

WolframAlpha was able to process 108 out of the 120 questions in the dataset, achieving a global precision of 0.82 and a precision over processed questions of

[7] https://www.wolframalpha.com/.

Table 2. Experimental results for the fine-grained approach

	QA over CodeOntology	WolframAlpha
Number of questions	120	120
Processed questions	116	108
Correct answers	109	98
Precision (global)	0.91	0.82
Precision (processed questions)	0.94	0.91

0.91. On the other hand, our approach based on CodeOntology allows processing 116 questions and 109 of such items have been answered correctly. Hence, on this task, the implemented system outperforms WolframAlpha, reaching a precision over processed questions of 0.94.

Interestingly, we noticed that WolframAlpha fails in computing the correct result for some simple queries, as shown in Table 3.

Table 3. Results obtained by WolframAlpha on a set of simple queries

WolframAlpha		
Input	Interpretation	Result
Add 2 to 4	$2 + 4$	6
Add 2 to the max between 3 and 4	$2^{\max\{3,4\}}$	16
Add 2 to the sum of 1 and 3	2^{1+3}	16
What is the uppercase of “abc”?	ToUpperCase[“abc”]	"ABC"
Convert “abc” to uppercase	ToUpperCase[“Convert \“abc\” to”]	"CONVERT\"ABC\"TO"
What is the length of “abcd”?	StringLength[“abcd”]	4
Sum 1 to the length of “string”	-	-

For instance, despite the system is capable of correctly interpreting commands like *“Add 2 to 4”*, it does not parse successfully slightly more complicated sentences such as *“Add 2 to the max between 3 and 4”*.

On the other hand, our approach is able to process correctly the same queries, as shown in Table 4.

Moreover, we can classify questions depending on whether both the systems, only one of them or none of them was able to provide the correct answer. Such categorization is shown in Table 5.

We can use the values reported in Table 5 to perform a McNemar exact test by comparing the case where the two systems provide discordant results (b and c), to a binomial distribution with size parameter $n = b + c$ and $p = 0.5$. The test shows that there exists a statistically significant difference between the two systems, with a confidence level of 99.8%.

Table 4. Results obtained by our approach on a set of simple queries

AskCO		
Input	Interpretation	Result
Add 2 to 4	`Math.addExact(2,4)`	6
Add 2 to the max between 3 and 4	`Math.addExact(2,Math.max(3,4))`	6
Add 2 to the sum of 1 and 3	`Math.addExact(2,Integer.sum(1,3))`	6
What is the uppercase of "abc"?	`"abc".toUpperCase()`	`"ABC"`
Convert "abc" to uppercase	`"abc".toUpperCase()`	`"ABC"`
What is the length of "abcd"?	`"abcd".length()`	4
Sum 1 to the length of "string"	`Long.sum(1,"string".length())`	7

Table 5. Comparison between AskCO and WolframAlpha

	WolframAlpha (correct)	WolframAlpha (failed)	
AskCO (Correct)	$a = 97$	$b = 12$	109
AskCO (Failed)	$c = 1$	$d = 10$	11
	98	22	120

6 Conclusion

This paper introduces two approaches for answering end-user questions on the execution of Java methods. On the one hand, our coarse-grained approach only allows mapping natural language commands to the execution of a single method, but it supports arguments of any arbitrary type, including user-defined classes. On the other hand, the fine-grained approach can handle more complex questions, possibly requiring the execution of multiple methods. However, the input of this approach is a single natural language question which includes the actual arguments as literals, thereby limiting the kinds of the parameters that can be passed by the user. Overall, experimental results show that the approach is promising and, subsequently, it can be effectively used for semantic code search and reuse over CodeOntology.

References

1. Atzeni, M., Atzori, M.: CodeOntology: RDF-ization of source code. In: d'Amato, C., et al. (eds.) ISWC 2017. LNCS, vol. 10588, pp. 20–28. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68204-4_2
2. Atzeni, M., Atzori, M.: CodeOntology: querying source code in a semantic framework. In: 16th International Semantic Web Conference (Posters & Demo) (2017)
3. Atzeni, M., Atzori, M.: CodeOntology OpenJDK8 dataset. Figshare (2017). https://doi.org/10.6084/m9.figshare.5234878
4. Atzori, M., Mazzeo, G.M., Zaniolo, C.: QA^3: a natural language approach to question answering over RDF data cubes. Semant. Web J. (2018)

5. Devanbu, P.T., Brachman, R.J., Selfridge, P.G., Ballard, B.W.: Lassie - a knowledge-based software information system. In: 12th International Conference on Software Engineering, pp. 249–261. IEEE Computer Society Press (1990)
6. Würsch, M., Ghezzi, G., Reif, G., Gall, H.C.: Supporting developers with natural language queries. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering, ICSE 2010, vol. 1, pp. 165–174. ACM (2010)
7. Bernstein, A., Kaufmann, E., Kaiser, C., Kiefer, C.: Ginseng: a guided input natural language search engine for querying ontologies. In: 2006 Jena User Conference, Bristol , UK (2006)
8. Panchenko, O., Mller, S., Plattner, H., Zeier, P.D.A.: Querying source code using a controlled natural language. In: The Sixth International Conference on Software Engineering Advances, ICSEA 2011, pp. 369–373, June 2011
9. Kimmig, M., Monperrus, M., Mezini, M.: Querying source code with natural language. In: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011) (2011)
10. Liu, H., Lieberman, H.: Metafor: visualizing stories as code. In: 10th International Conference on Intelligent User Interfaces, IUI. pp. 305–307 (2005)
11. Liu, H., Lieberman, H.: Programmatic semantics for natural language interfaces. In: Extended Abstracts on Human Factors in Computing Systems. CHI (2005)
12. Sales, J.E., Handschuh, S., Freitas, A.: SemEval-2017 task 11: end-user development using natural language. In: 11th International Workshop on Semantic Evaluation, pp. 556–564 (2017)
13. Atzeni, M., Atzori, M.: Towards semantic approaches for general-purpose end-user development. In: 2nd IEEE International Conference on Robotic Computing, IRC, pp. 369–376 (2018)
14. Quirk, C., Mooney, R., Galley, M.: Language to code: learning semantic parsers for if-this-then-that recipes. In: 53rd Annual Meeting of the Association for Computational Linguistics, ACL, pp. 878–888 (2015)
15. Lehmann, J.: DBpedia - a large-scale, multilingual knowledge base extracted from Wikipedia. Semant. Web J. **6**(2), 167–195 (2015)
16. Ferragina, P., Scaiella, U.: TAGME: on-the-fly annotation of short text fragments (by Wikipedia entities). In: 19th ACM International Conference on Information and Knowledge Management, CIKM, pp. 1625–1628 (2010)
17. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Advances in Neural Information Processing Systems, pp. 3111–3119. Curran Associates, Inc. (2013)
18. Manning, C.D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S.J., McClosky, D.: The stanford CoreNLP natural language processing toolkit. In: Association for Computational Linguistics (ACL) System Demonstrations, pp. 55–60 (2014)