Ontology-Based Determining of Evaluation Order of C Expressions and the Fault Reason for Incorrect Answers[⋆]

Oleg Sychev[1][0000-0002-7296-2538] and Nikita Penskoy[1][0000-0002-4443-33994]

Volgograd State Technical University, Lenin Ave, 28, Volgograd, 400005, Russia
oasychev@gmail.com

Abstract. The article describes the process of developing an automated tool to determine the evaluation order in the expression on C programming language and describe the faults made by a student in this task. The main part of this tool is based on the SWRL rule set, which can be divided into several different tasks: calculation of the evaluation order of expression based on constructing abstract syntax tree of expression, determine contradictions with the partial student's evaluation order, and describing fault reasons.

Keywords: ontology · expression evaluation order · C programming language · SWRL.

1 Introduction

Developing basic programming skills is a high-demand area of teaching that is challenging both for teachers and students that makes it a popular research area. Introductory programming courses are a good example of a situation where students have to learn a significant number of new concepts but these concepts are formal and well-defined. This makes basic programming courses a good area to test different forms of concept-based education methods, including using ontologies.

Ontologies is commonly used to representing mathematical knowledge as shown in [4], can be used to model the certain domain [2], interactively help learners understand the new domain [8] or new code [7]. On the other hand, there are code analyzers that are not aimed at learning programming, but creating a code representation in the model; some of them are based on the ontological approach, like the static ontological code analyzers [9, 6].

In order to facilitate the comprehension of concepts, ontologies should model the usage of these concepts in the typical domain tasks. This allows calculating the correct answer for a question and, with the student's response converted to

⋆ The reported study was funded by RFBR, project number 20-07-00764.

the ontology, determining the exact fault reason that allows explanatory feedback on mistakes. Such ontologies would have an advantage over procedural task-solving systems which have limited capability for explanatory feedback. However, the suitability of current ontological reasoners for solving problem-oriented tasks needs to be researched first. In this study, we aimed at creating ontology to determine the evaluation order of a C-language expression and detect fault reasons in student-supplied evaluation order using SWRL rules. The studied concepts were operators, their precedence and associativity, sequence points, and order of evaluation.

The order of evaluation in C programming language is a partial order. Because of sequenced-before relation, evaluation order isn't fully defined by the abstract syntax tree of the expression but can be expressed as a directed acyclic graph where child nodes must be evaluated before the parent nodes.

$$a \ + \ b \ * \ c \ < \ f(d \ ? \ e \ || \ \sim f \ : \ j \ + \ --k, \ l) \tag{1}$$

Fig. 1 shows the evaluation order graph for expression 1, the purple arrow pointing to the box shows that `d` should be calculated earlier than all the tokens in the box because of the ternary operator.

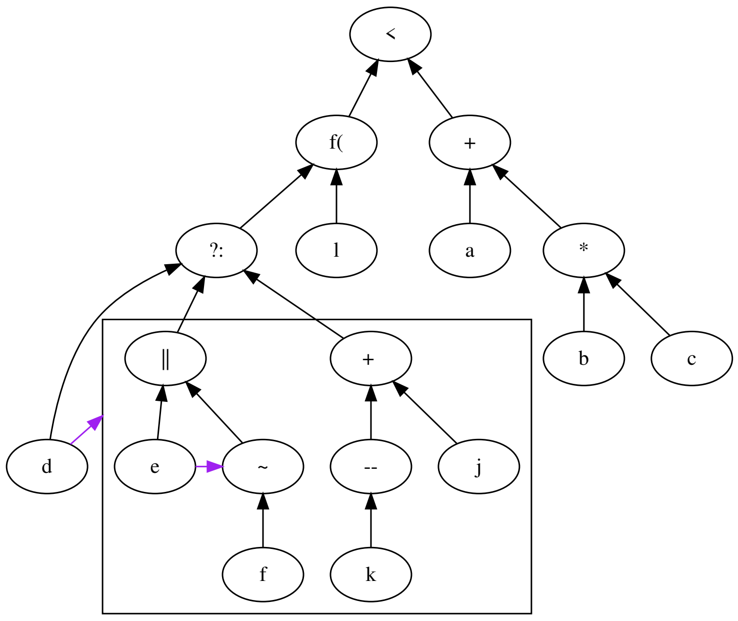

Fig. 1. Evaluation order graph example

So, for the questions about evaluation order, the developed ontology should determine the sequenced-before relation as a direct acyclic graph, compare the student's evaluation order marks with the graph to determine fault reasons, and provide information necessary to generate explanatory feedback.

2 Determining order of evaluation

The ontology model receives an expression split into tokens. For each token, its position in the expression and the text of the token are set. Each input token is representing by several individuals in ontology, one for each step of constructing abstract syntax tree (AST). First, the ontology builds the AST for the given expression. This takes place inside the ontology that records the state of the expression at each step, marking which operators are evaluated and which evaluation results have already been used by the other operators. Using SWRL rules for building AST differs from regular imperative algorithms and supplies the later fault-reason determining stage with the necessary information.

A tree node has data properties `arity`, `precedence`, `associativity`, `prefix_postfix` (for unary operators) filled at initial step of constructing AST using information from text and neighboring tokens. At each step of constructing AST, a set of SWRL rules finds an operator that can be evaluated and generates the tree copy for the next step, connecting the evaluated operator with its operands using property `has_operand` and changing markers of evaluated and used operators.

To create the evaluation-order graph based on AST, the rules determining sequenced-before relations are applied, accounting for the operators guaranteeing a certain order of evaluation for their operands like binary logical operators, the ternary operator, and the comma operator.

After the end of this stage, AST and the evaluation order graph are copied to a special step that will be used to perform further calculations while the steps for constructing the tree remain to provide information for generating a detailed explanation of found errors.

3 Determining fault reasons

For fault reason determining, students must provide their responses, assigning numbers to the operators in their evaluation order. As the evaluation order is a partial order, many expressions allow several correct answers. The nodes lacking student's order marks are assumed to be evaluated after all marked nodes so they are assigned large numbers corresponding to their positions in the graph. A student makes a mistake by assigning a smaller number to a parent node of the evaluation order graph than to its child node both for the edges representing operands and sequenced-before relations.

The developed ontology provides information about all faults in the given student's answer, including fault reasons and supplementary information for generating the explanation message using a natural-language template. All possible faults are split into two categories: calculating the result of an operator before its operand (for example, mistakes in operator precedence or associativity) and wrong evaluation order of two operators caused by sequence points created by another operator (e.g. trying to evaluate the right operand of a logical `and` operator before finishing evaluating its left operand). Errors is stored as object properties between two error nodes named like `student_error_strict_operands_order` or `student_error_more_priority_left`, etc. In the following examples, the \$ followed by a number shows the position of the preceding operator in the student's evaluation order. For example, formula 2 demonstrates a simple mistake caused by the student ignoring associativity. The mistake message generated for it will be: "Operator + on pos 4 cannot be evaluated before operator + on pos 2 because operator + has left associativity".

$$a\ +\$2\ b\ +\$1\ c \tag{2}$$

Formula 3 shows a more complex mistake involving three operators. The mistake message for it will be: "Operator + on pos 6 that belongs to the right operand of operator || on pos 4 cannot be evaluated before its left operand, including operator * on pos 2."

$$a\ *\$2\ b\ ||\$3\ c\ +\$1\ d \tag{3}$$

So the developed ontology can detect fault reason for mistakes in operators precedence, associativity, and sequence points during evaluating an expression, providing the necessary information for explanatory feedback including the fault reason and operator nodes.

The ontology was tested using Java unit testing framework; constructing expression AST (144 tests), evaluation order tree (10 tests), and determining fault reasons (20 tests, including mistake combination) were tested separately. The ontology and testing system is available on Github[1].

4 Conclusion and further work

We developed ontology able to solve determine the evaluation order of a C-language expression and fault reasons in a student's answer with the potential to generate explanatory feedback including correctly processing multiple errors. It also supports a subset of C++ language. The results prove that ontological reasoners are capable to process expressions and determine fault reasons so they can be used as a basis of the testing system generating questions, solving them, and generating explanatory feedback for students.

The developed ontology differs from other ontologies used to generate and grade questions [5] in that it captures domain knowledge about concept usage in the form of SWRL rules and provides information for the generation of explanatory feedback based on the domain laws that the student violated in their answer. Generating feedback is described as an important field for further research in recent question-generation survey [3]. Our ontology can be used for

[1] https://github.com/ShadowGorn/ontology_evaluation_order_check

grading and providing feedback for complex expression-related tasks like determining an order of evaluation for randomly-generated expression which is more complex questions than currently existing systems, allowing extending generated questions to higher levels of Bloom's taxonomy [1].

The further development of the ontology includes fully supporting C++ expressions containing complex operators and operands like template function calls and initializer lists and supporting other programming languages like Python, adapting the ontology to the chosen language by tagging its rules. The ontology also will be enhanced to support other expression-related exercises, including determining subexpression data types for statically-typed languages. To use the developed ontology in the learning process, we will develop the core of the testing system and question-generation routines.

The developed testing system will be modular and able to work with other domains provided that the relevant ontologies and question-generation routines are implemented. Our research group is open for collaboration with researchers interested in developing support for their domains.

References

1. Anikin, A., Sychev, O.: Ontology-based modelling for learning on bloom's taxonomy comprehension level. In: Biologically Inspired Cognitive Architectures Meeting. pp. 22–27. Springer (2019)
2. Grévisse, C., Botev, J., Rothkugel, S.: An extensible and lightweight modular ontology for programming education. In: Colombian Conference on Computing. pp. 358–371. Springer (2017)
3. Kurdi, G., Leo, J., Parsia, B., Sattler, U., Al-Emari, S.: A systematic review of automatic question generation for educational purposes. International Journal of Artificial Intelligence in Education **30**(1), 121–204 (2020)
4. Lange, C.: Ontologies and languages for representing mathematical knowledge on the semantic web. Semantic Web **4**(2), 119–158 (2013)
5. Leo, J., Kurdi, G., Matentzoglu, N., Parsia, B., Sattler, U., Forge, S., Donato, G., Dowling, W.: Ontology-based generation of medical, multi-term mcqs. International Journal of Artificial Intelligence in Education **29**(2), 145–188 (2019)
6. Pattipati, D.K., Nasre, R., Puligundla, S.K.: OPAL: An extensible framework for ontology-based program analysis. Software: Practice and Experience **50**(8), 1425–1462 (Mar 2020)
7. Rawashdeh, M., Alnusair, A., Almiani, M., Sawalha, L.: Jmentor: An ontology-based framework for software understanding and reuse. In: 2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA). pp. 1–8. IEEE (2019)
8. Silva, V., Dorça, F.: An automatic and intelligent approach for supporting teaching and learning of software engineering considering design smells in object-oriented programming. In: 2019 IEEE 19th International Conference on Advanced Learning Technologies (ICALT). vol. 2161, pp. 321–323. IEEE (2019)
9. Zhao, Y., Chen, G., Liao, C., Shen, X.: Towards ontology-based program analysis. In: 30th European Conference on Object-Oriented Programming (ECOOP 2016). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)