Title

Anonymous Author(s)

Abstract

Abstract

1 Introduction

Automatic code analysis tools are commonly used in educational settings to support the evaluation of programming assignments. While many existing tools focus on correctness or performance, they often overlook the importance of how students solve the given problems, particularly whether they follow the concepts and constructs introduced during the course. In introductory programming courses, it is important not only to assess whether a student's solution is correct, but also to ensure that the student is applying the concepts and techniques taught in class. Automatic code analysis can help detect when students rely on language features or external solutions that bypass the intended learning objectives. To adress this need, we propose a multi-language code analysis tool based on Parsing Expression Grammars (PEGs), designed to detect the use of advanced or unauthorized constructs in students' code submissions. This tool aims to assist educators in enforcing pedagogical boundaries while maintaining flexibility across different programming languages.

2 Related work

Atkinson and Griswold[1] presents the matching tool TAWK, which extends extend the pattern syntax of AWK to support matching of abstract syntax trees. In TAWK, pattern syntax is language-independent, based on abstract tree patterns, and each pattern can have associated actions, which are written in C for generality, familirity and performance. Throughout the paper, a prototypical example of extracting a call-graph from a given code, giving examples in differents tools for pattern matching. At a later section, we also present an extraction of a call-graph using the tool developed in this paper.

Kopell et al.[3] presents an approach for building source-to-source transformation that can run on multiple programming languages, based on a representation called incremental parametric syntax (IPS). In IPS, languages are represented using a mixture of language-specific and generic parts. Transformations deal only with the generic fragments, but the implementer starts with a pre-existing normal syntax definition, and only does enough up-front work to redefine a small fraction of a language in terms of these generic fragments. The IPS was implemented in a Haskell framework called *Cubix*, and currentl supports C, Java, JavaScript, Lua, and Python. They also demonstrate a whole-program refactoring for threading variables through chains of function calls and three smaller source-to-source transformations, being a hoisting transformation, a test-coverage transformation and a the three-address code transformation.

Premtoon et al.[5] presents a tool called *Yogo*, that uses an approach to semantic code search based on equational reasoning, that considers not only the dataflow graph of a function, but also the dataflow graphs of all equivalents functions reachable via a set of rewrite rules. The tool is capable of recognizing differents variations of the same operation and also when code is an instance of a higher-level concept. *Yogo* is built on the *Cubix* multi-language infraestructure and can find equivalent code in multiple languages from a single query.

Silva et al.[6] proposes *RefDiff 2.0*, a multi-language refactoring detection tool. Their approach introduces a refactoring detection algorithm that relies on the Code Structure Tree (CST), a representation of the source code that abstract away the specificities of particular programming languages. The tool has results that are on par with state-of-the-art refactoring detection approaches specialized in the Java language and has support for two other popular programming languages: JavaScript and C, demonstrating that the tool can be a viable alternative for multi-language refactoring research and in practical applications of refactoring detection.

van Tonder and Le Goues[7] proposes that the problem of automatically transforming programs can be decomposed such that a common grammar expresses the central context-free language (CLF) properties shared by many contemporary languages and open extensions points in the grammar allow customizing syntax and hooks in smaller parsers to handle language-specific syntax, such as comments. The decomposition is made using a Parser Parser combinator (PPC), a mechanism that generates parsers for matching syntactic fragments in source code by parsing declarative user-supplied templates. This allows to detach from translating input programs to any particular abstract syntax tree representation, and lifts syntax rewriting to a modularly-defined parsing problem. They also evaluated *Comby*, an implementation of the approach process using PPC, on a large scale multi-language rewriting across 12 languages, and validated effectiveness of the approach by producing correct and desirable lightweight transformations on popular real-world projects.

Matute et al.[4] proposes a search architecture that relies only on tokenizing a query, introducing a new language and matching algorithm to support tree-aware wildcards by building on tree automata. They also present *stsearch*, a syntactic search tool leveraging their approach, which supports syntactic search even for previously unparsable queries.

Ierusalimschy [2] proposes the use of PEGs as a basis for text pattern-mathing and presents LPEG, a pattern-matching tool based on PEGs for the Lua scripting language, and a Parsing Machine that allows a small and efficient implementation of PEGs for pattern matching. This allow LPEG to have both the expressive power of PEGs with the ease of use of regular expressions. LPEG also seems specially suited for languages that are too complex for traditional pattern-matching tools but do not need a complex yacc-lex implementation, like domain-specific languages such as SQL and regular expressions, and even XML.

3 References

References

[1] Darren Atkinson and William Griswold. Effective pattern matching of source code using abstract syntax patterns. *Softw., Pract. Exper.*, 36:413–447, 04 2006.

[2] Roberto Ierusalimschy. A text pattern-matching tool based on parsing expression grammars. *Softw. Pract. Exper.*, 39(3):221–258, March 2009.

[3] James Koppel, Varot Premtoon, and Armando Solar-Lezama. One tool, many languages: language-parametric transformation with incremental parametric syntax. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[4] Gabriel Matute, Wode Ni, Titus Barik, Alvin Cheung, and Sarah E. Chasins. Syntactic code search with sequence-to-tree matching: Supporting syntactic search with incomplete code fragments. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.

[5] Varot Premtoon, James Koppel, and Armando Solar-Lezama. Semantic code search via equational reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 1066–1082, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Danilo Silva, João Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, 47(12):2786–2802, Dec 2021.

[7] Rijnard van Tonder and Claire Le Goues. Lightweight multi-language syntax transformation with parser parser combinators. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 363–378, New York, NY, USA, 2019. Association for Computing Machinery.