Using abstract lexical analysis and parsing to detect errors in string-embedded DSL statements

Andrey Breslav  
St. Petersburg State University of IT, Mechanics and Optics  
Email: abreslav@gmail.com

Aivar Annamaa  
University of Tartu  
Email: aivar79@ut.ee

Varmo Vene  
University of Tartu  
Email: varmo@cs.ut.ee

Introduction. String-embedding of domain-specific languages (DSLs) is a widely adopted approach, according to which, statements of a DSL are written in string literals inside a program in a general-purpose language. For example, the standard Java Database Connectivity (JDBC) API takes this approach to let Java programs interact with database engines by executing SQL queries, which are passed in as plain strings. Other examples include server-side programs generating HTML pages (e.g., JSP) and running dynamically-typed programs from within programs written in a statically typed language (e.g., Java Scripting API). All these applications have one inconvenience in common: the DSL statements inside the string literals are not checked at compile-time, and even simple mistakes like misspelled names or missing parentheses cause run-time failures, which affects developers’ productivity.

We employ program analysis techniques to statically detect syntactical errors in string-embedded DSL statements. Our analysis has a modular structure that separates the following phases:

- collecting information about the strings constructed by a program,
- performing lexical analysis for these strings,
- performing syntactic analysis.

Collecting abstract strings. The first stage of the analysis is detecting the embedded DSL statements among all the string literals in the program. These strings are passed as arguments to API functions which are known to accept DSL statements (e.g., JDBC functions expect SQL statements in string arguments). An expression which is an argument for such an API call is called a hotspot. For each hotspot, a program slice is obtained and path-insensitive constant propagation is performed in order to construct a set of all possible values the hotspot may be evaluated to. This set is represented as a regular expression over the alphabet of individual characters and is referred to as an abstract string. Further stages of the analysis solve the conventional problems of lexical and syntactical analysis, but for the case of abstract strings instead of plain text.

Abstract lexical analysis. Conventional lexical analysis is a process of transforming a finite string of characters into a string of tokens. This process can be formalized as a finite-state transduction of the input string. In our case, the input is not a single string, but a set of strings, which may be infinite, if the initial program used loops to construct string values. Thus, the purpose of the lexical analysis is to break every string in this set into tokens. The abstract strings are represented as regular expressions over the character alphabet, which can be transformed into a finite automaton (FA) by the conventional algorithm [4]. As a finite-state transduction of a regular language is itself regular [2], we can apply a finite-state transducer (FST) generated by a standard tool like JFlex [5] to the whole input automaton without any precision loss. We can refer to this procedure as abstract lexical analysis; it builds a FA over the alphabet of tokens, that forms the input for the parsing phase.
Abstract syntactical analysis. For syntactical analysis of abstract strings we employ the technique proposed by Doh et al. [3] and referred to as abstract parsing. The original technique is based on the conventional LALR-parsing (as done by YACC or Bison [1]), but over the character alphabet; we extend it to integrate with lexical analysis and to handle ambiguous grammars. For the latter, we utilize the same approach as the GLR algorithm [8].

The central idea of abstract parsing is based on the fact that the state of an LR-parser is totally characterized by (a) its action tables, (b) the current position in the input text, and (c) the current stack of states (the top state is the current one). When, instead of a plain string, the input is a FA over the alphabet of tokens, a state of this FA plays the role of the current position, the action tables are unchanged, so the goal is to maintain for each state all possible stacks that can appear at this position. The stacks are constructed by a fix-point computation that performs parsing actions according to the tokens generated by the transitions of the input automaton.

Since the problem of inclusion of a regular language (lexer output) into a context-free one (e.g., SQL) is undecidable [4], a fix-point may not be reached on some inputs. Thus, we need to introduce abstraction. After comparing the ones proposed in [3] and [6], we chose the latter for it is far more adequate. This abstraction is bounding the depth of the stacks by a constant, which makes the set of possible stacks finite (since they are comprised of states which in turn form a finite set).

In practice, the publicly available grammars of widely-used DSLs, such as SQL grammar from ISO, are written with only documenting purpose in mind, hence they do not belong to any convenient class (such as LALR(1)) and contain many ambiguities. In addition, the size of such a grammar usually does not allow manual disambiguation in any reasonable time. Thus, there is a strong need to support ambiguous grammars in our setting. To do this, we replaced the conventional parsing stacks of LALR with multi-stacks inspired by the GLR algorithm [8]. Since a parsing table for an ambiguous grammar may have many actions for the same parsing state and input token, there may be many stacks yielded by one step of the parser, some of which will later be rejected because they lead to an error state. A multi-stack stores all these options and supports rejection upon reaching an error state. If all the options in a multi-stack were rejected, a syntactic error is detected.

Discussion and future work. We have presented a program analysis for string-embedded DSLs, comprised of three stages: (a) collecting abstract strings, (b) abstract lexical analysis and (c) abstract parsing. Our work uses the same principles as [3] and [6], but differs from these developments in the following ways: (1) it is modular: the analysis phases are clearly separated, which serves for reuse and easier detection of the causes of false alarms; (2) it uses explicit lexical analysis, where the existing approaches work directly on the character alphabet, which is considered rather inconvenient [7]; the lexical analysis does not introduce any loss of precision; (3) it supports ambiguous grammars, which, together with lexical analysis, facilitates the use of existing grammar definitions, as opposed to defining huge grammars manually.

Our approach is implemented as a plug-in for the Eclipse platform which currently analyzes SQL statements embedded into Java programs. It is integrated into the incremental building infrastructure and is suitable for interactive use, similarly to the Java compiler used by Eclipse.

As the possible directions for the future work, we consider supporting basic semantic analysis, such as detecting roles of identifiers (table name, column name etc.), as well as “abstract error recovery” to support incomplete queries. These capabilities will facilitate code completion which is widely acknowledged as one of the most convenient features of modern IDEs, but is not available for embedded DSL statements yet.
Abstract lexical analysis and parsing  A. Breslav, A. Annamaa, V. Vene

Acknowledgements. This work was partially supported by the Estonian Science Foundation grant #8421, as well as the Software Technology and Applications Competence Center (STACC), and the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS.

References