Comparison of Source Code Transformation Methods for Clang

Máté Karácsony

Clang is a robust, industrial strength C, C++ and Objective-C compiler. Its library based architecture made it a common platform for custom tools which are analyzing and transforming programs. As it provides a rich abstract syntax tree representation, it is widely used especially for source-to-source transformations, like vectorization of loops [1], or translation between different language dialects [2]. Despite the presence of these tools, Clang currently does not have a complete and uniform library to support specifically source-to-source transformations.

Conventional transformation engines are working by manipulating the abstract syntax tree (AST) of the input source code, which is generated by a parser. A particular transformation is usually described by a set of rules. Application of a single rule is performed by replacing or mutating matching AST nodes. These rules are applied according to a strategy to create a modified AST. The output of the transformation process is computed by pretty-printing this structure into its source code representation [3].

Clang provides different methods to traverse and match specific parts of AST, but it does not provide an interface to define and apply transformation rules in a straightforward way. While its AST is designed to be immutable, classical AST-based transformations are simply not eligible. Moreover, pretty-printing the whole transformed syntax tree is not practical in the case of preprocessed languages, since all macros will be expanded in the output, which impedes further code maintenance. For these reasons, Clang incorporates a simple source code rewriting facility, that is able to replace source text in the given ranges. This low-level API can be used to safely change small text fragments in input source files, but it is not able to handle complex cases.

As these built-in transformation capabilities are limited, every tool implementation created its own procedures and utilities for this purpose. In this paper we will present and compare various existing transformation techniques, including the built-in options and custom solutions found in Clang-based tools.

Three key aspects of these methods will be examined: expressiveness, implementation cost and reusability. Expressiveness shows what kind of transformations can be described, and how these are represented. The second aspect measures the effort needed to implement the given technique. Finally, reusability expresses how easy is to apply the specific method to achieve different kinds of transformations.

References

[1] Krzikalla, Olaf, et al. "Scout: a source-to-source transformator for SIMD-Optimizations." Euro-Par 2011: Parallel Processing Workshops. Springer Berlin Heidelberg, 2012.

[2] Martinez, Gabriel, Mark Gardner, and Wu-chun Feng. "CU2CL: A CUDA-to-OpenCL translator for multi-and many-core architectures." Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on. IEEE, 2011.

[3] Van Wijngaarden, Jonne, and Eelco Visser. "Program transformation mechanics. a classification of mechanisms for program transformation with a survey of existing transformation systems." Technical report UU-CS 2003-048 (2003).