A Reversing Technique for Symbol Table Verification on Compiler Constructions

Yunsik Son\textsuperscript{1}, Seman Oh\textsuperscript{1}, Yangsun Lee\textsuperscript{2}

\textsuperscript{1}Dept. of Computer Engineering, Dongguk University
26 3-Ga Phil-Dong, Jung-Gu, Seoul 100-715, KOREA
\{sonbug, smoh\}@dongguk.edu

\textsuperscript{2}Dept. of Computer Engineering, Seokyeong University
16-1 Jungneung-Dong, Sungbuk-Ku, Seoul 136-704, KOREA
Corresponding Author: ysllee@skuniv.ac.kr

Abstract. In this paper, we will deal with the reversing technique to verifying the symbol table in Objective C compiler. The proposed technique reconstructs inputted Objective C program segments - declarations part - by using information of the symbol table inserted during the process of declaration analysis phase of the Objective C compiler, and therefore we can verify the completeness of symbol table design and validity of information inserted in the symbol table.

Keywords: Compiler Construction, Symbol Table, Reverse Engineering, De-compiler

1 Introduction

Symbol table is an essential module in the compiler constructions that is used on lexical analysis, syntax analysis, semantic analysis and code generation. Especially, semantic analysis phase is core part that is analyzing semantics of expressions and statements, and verifying symbols.

But, the existing Objective C compilers are designed to translate source programs into target codes and then execute them. Also, this translation method needs source programs to generating target codes for each platform, and reusability and portability of codes will also decrease because target codes have to be different for each platform they are run under [1]. For that reason, much research is taking place in various fields in an effort to develop a retargetable compiler and virtual machine that execute application programs without recompiling or modifying them though processor or operating systems are changed.

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT and future Planning(No.2013R1A2A2A01067205).

ISSN: 2287-1233 ASTL
Copyright © 2014 SERSC
In this paper, we introduce the reversing technique that rebuilds the attributes inputted into the symbol table to the original program in order to verify and analyze the symbol table designed for the Objective C compiler in development phase.

2 Symbol Table

The symbol table is a data structure to be used to keep and manage track of scope and hiding information about identifiers. Lexical and syntax analysis takes place in the compiler, and the abstract syntax tree generated by the SDT (Syntax-directed Translation) is analyzed to collect and reference the attributes of the recognized names. These attributes are inserted in a symbol table where the names are defined and the validity of the use of these names and attributes is verified in the semantic analysis phase. After this, the code generation phase produces the correct code using the attributes [2].

The symbol table designed in the Objective C compiler divided into window table and sub-tables. Window tables consist of symbol table, type table. And sub-tables consist of a concrete table, an abstract table, an aggregate table, a member table. Figure 1 shows the relationships between the composed symbol tables and each table.

![Fig. 1. Relationship of Symbol Table for Objective C Compiler](image)

3 Reversing Technique for Symbol Table

The reverse translator inserts the declaration part extracted from Objective C (*.m) files into the symbol table and uses the information to restore them to the Objective C declaration part. The reverse translation is mainly subcategorized into a symbol process and type process. It distinguishes the tables that need to be translated reversely by sequentially searching the symbol table and the type table. Moreover, the sub table is
referenced by the table index and the information is used to translate reversely to an Objective C program.

The symbol process is the part that translates reversely all the information corresponding to the external variables in the symbol table. The extracted information details the variable type through the concrete process and determines, through this phase, if it is an array, a function, or a regular variable, and then outputs it after translating reversely it to an Objective C program. A parameter process is added for functions with parameters, and the parameters to be translated reversely are outputted. Figure 2 shows the flow of the symbol process.

![Flow of the Symbol Process](image)

Fig. 2. Flow of the Symbol Process

The type process is composed of a member process, a link process, the abstract process and the parameter process. It searches the user-defined type information in the type table to classify the information into class, interface, protocol, struct, union, enum, namespace and typedef for a reverse translation. The member process is performed when the type is class, interface, protocol, struct and union, and is similar to the symbol process in that it outputs after translating reversely the member variables and member functions included in the type through the abstract process and the parameter process. Figure 3 is the flow of the type process.

![Flow of the Type Process](image)

Fig. 3. Flow of the Type Process

The following table 1 is example programs that are inserted the attributes of the declarations into a symbol table and translates reversely in order to restore it back to an Objective C programs.

<table>
<thead>
<tr>
<th>Original type</th>
<th>@interface NSObject @property NSString *myName;</th>
</tr>
</thead>
</table>

Table 1. Experimental Results of Symbol Table Reversing
4 Conclusion

In this paper, design and implementation of a reverse translator that verifies and analyzes the symbol table designed during the development stage of the Objective-C compiler. The reverse translator’s role is to restore, using only the information in the symbol table, back to the original program. Therefore, it is possible to analyze and verify the completeness of the designed symbol table and the information on the identifiers stored in the symbol table. Moreover, based on the verified symbol table, a correct code can be generated by examining the use of the referenced identifiers and the attributes of the stored identifiers in the code generation stage.

References