

Compiling C++ Programs to Java Bytecode

Gongzhu Hu and Avinashkumar Gadapa
Department of Computer Science
Central Michigan University
Mount Pleasant, MI 48859, USA
hu@cps.cmich.edu

Abstract

It is very desirable to run programs on a variety of platforms. The only programs today that can run on different platforms are those written in Java. Although methods have been developed to allow cross-language applications, these applications are still mostly be hardware and/or operating system platform dependent. In this paper, we describe a platform-neutral compiler for a C++ like language that generates Java bytecode to run on any platform where a Java Runtime is available.

1. Introduction

The existing and commonly used compilers fall into two categories:

- (1) for a specific programming language and a specific hardware,
- (2) for a specific programming language but platform neutral.

Most compilers are in category (1). For example, a C++ compiler on a UltraSparc workstation compiles C++ programs and generate executable code that runs only on the UltraSparc CPU. A Fortran compiler on an Intel Pentium machine compiles Fortran programs that runs only on the Intel Pentium CPU.

In order to be platform neutral for compilers in category (2), some sort of “virtual machine” must be available on the platform that bridges the gap between the code generated by the compiler and the code that can be executed on the hardware. Java compiler is a good example. This platform-neutral feature is very attractive for many applications that run on a variety of hardware platforms. However, the applications must be written in Java.

The question now is: Can we have a platform-neutral compiler for a language other than Java? That is, can we compile, say, a C++ program that also runs on different platforms? This paper is to address this question and

present a design of such a compiler. There have been some work done in this area, but most were either doing the language conversion at the source level or at binary level. We haven't seen a non-Java compiler that directly translate from source program to “executable” code that runs on different platforms.

In this paper, we describe a compiler for a smaller-scale C++-like language that generates Java bytecode to run on any platform where a Java Runtime (or Java Virtual Machine, JVM) is available. The compiler is just like other classic compilers – It performs lexical, syntax, and semantic analysis, and generates code for a target machine. Differs from other traditional compilers, the target machine for our C++ compiler is not for a specific hardware; rather, the target machine is the Java Virtual Machine.

2. Related work

Researchers and developers have proposed and developed various language processing methods to facilitate cross-language applications. Most of these methods either perform language conversion at source level or at a lower level. The source level conversion methods translate a program written in language L_1 to an equivalent program in language L_2 . Lower level conversions deal with intermediate code mapping.

In [11], Terekhov and Verhoef provided examples that dealt mostly with COBOL but apply to many instances of language conversions. They argued that the difficulties of such conversions were underestimated and manifold. The important facts they articulated about source conversion include emphasis on reliability of the conversion process, expertise in source and target languages, a well designed conversion system, and difficulty in going from a rich source language to a minimum target language.

Malton stated that there are several ad-hoc techniques for source conversion but only few systematic approaches [7]. He defined a set of goals for source conversion and identified three different conversion styles which are

Dialect Conversion, API Migration, and Language Migration. Malton's observations were based on dialect conversions in COBOL, PL/I, and RPG domains, and pilot studies in source conversion from COBOL to Java, RPG to COBOL, and SQL to SQLj.

Kontogiannis [4] reported on the conversion of the IBM compiler back-end from a PL/I derivative to C++, Yasumatsu [14] discussed translating Smalltalk programs to C, Terekhov [10] described automated language conversion from a proprietary language to Visual Basic and COBOL. Hu and Hill [3] developed an assembly-like intermediate code to Java bytecode translation.

Frank Buddrus and Jörg Schödel developed a C++ to Java translator [2] at the source level. Tilevich [12] presented an automated C++ to Java transliteration tool that succeeded only in converting C++ constructs that have close Java equivalents. It is a significant help in the conversion of large volumes of legacy C++ code, but the software engineer has to do substantial manual work to finish the transliteration work.

Novosoft has released a C to Java transliteration tool [9] that has proven to translate large volumes of code correctly. However, their mapping of C data types to Java is non-intuitive and circumvents many Java security and run-time features.

C. Cifuentes et al in their Walkabout project developed a retargetable dynamic binary translation framework [13] based on the properties of retargetability and separation of machine-dependent from machine-independent concerns. It performs machine-level analysis to translate source binary code onto target binary code.

Microsoft's .Net also provides a virtual machine for cross-language applications, but it is only for a specific platform (Windows), which is what we would like to avoid.

3. Java virtual machine and Java bytecode

Here we briefly describe the Java Virtual Machine and the Java bytecode as the target language for our compiler.

The Java Virtual Machine is a component of the Java technology responsible for Java's cross-platform delivery [8]. It is an abstract computing machine that uses various memory areas and does not assume any particular implementation technology or host platform. It is not inherently interpreted, and it may be implemented by compiling its instruction set to that of a real CPU, as for a traditional programming language.

Although JVM was designed to support the Java programming language, it knows nothing about the Java. Instead it knows only of a particular file format, called the class file format. A class file contains JVM instructions or bytecodes, a symbol table, and other ancillary information. For security purpose the JVM imposes strong format and structural constraints on the code in a class

file. Any language with functionality that can be expressed in terms of a valid class file can be hosted by the JVM.

A JVM instruction consists of one-byte opcode specifying the operation to be performed followed by zero or more operands supplying data that are used by the operation. The JVM instruction set has 160 instructions with an upper bound of 256 imposed by the 8-bit representation. Most of these instructions correspond to the instructions used in register-based instruction sets while others are used to manipulate Java objects or invoke methods. Also most of the instructions are typed instructions since they operate on a set type of data.

Java bytecode is stored in a file called Java *ClassFile* that contains the virtual machine code for all the methods of the currently loaded class, a reference to the super class of the current class, all the fields that the class defines, the constant pool, and any other data required by the runtime system.

4. Design of our compiler

As mentioned before, our compiler consists of three main modules: lexical analysis (scanner), syntax analysis (parser), and semantic analysis. The parser is the driver module that controls the compilation process. Regular expressions are used to define the tokens in the language and context-free grammar in the form of BNF is used to define the syntax of the language. We implemented an LALR generator to create a parse table equivalent to a Characteristic Finite State Machine (CFSM) from the BNF. If no errors, the bytecode is written to a *.class* file, which will be ready for execution on any JVM.

4.1. LALR Parse Table Generator

We implemented a program called *parsegen* that takes a grammar in BNF as input and generates a LALR parse table. It uses a modified form of propagated lookahead algorithm [1, 5] to reduce the time required to generate the parse table. Instead of forming a propagate link in the Characteristic Finite State Machine (CFSM), it directly propagates the lookaheads to the next configuration. Spontaneous lookaheads are also handled as a part of the closure operation. The only links maintained are the one where the lookaheads are propagated after the next state is already constructed, and only if the next state's lookahead set has been changed.

There are only two ways a lookahead propagates from one configuration to the next, one during a shift operation on the configuration and the other on a closure operation on the state. Let S be a state being generated with configuration items (C_s) and lookahead symbols (L_s). Similarly, let P be a previously generated state with configuration items (C_p) and lookahead symbols (L_p). If C_s

= C_p , and $L_s \neq L_p$, then copy lookahead symbols in L_p which do not exist in L_s into L_s . Set flag in L_s indicating that it has acquired propagated lookaheads and C_p carries the link of L_s for further propagation of lookaheads in the future.

Another situation requiring a propagated link is a closure operation on another configuration. There are two sub cases to consider under this situation. The first case is when the lookaheads are computed directly from the FirstSet of the configuration, where the FirstSet does not include ϵ . These lookaheads are termed as spontaneous lookaheads because they do not depend on the configuration item from which they are calculated. The second case is where the FirstSet does include ϵ , in which the lookaheads of the configuration item from which the new configuration item is being formed are copied to the new configuration item. These lookaheads are propagated lookaheads and the old configuration item carries the link of the new configuration item for further lookahead propagation.

The main algorithms used in `parsegen` are shown below.

```
void parsegen() {
    CFSM cfsm = empty;
    Grammar grammar = loadGrammar(grammarDefinition);
    FirstSet firstSet = computeFirstSet(grammar);
    ConfigurationSet state;
    state.add(S → • α, { $ });
    state = closure(state);
    cfsm = computeTransitionStates(state);
    cfsm = propagateLookaheads(cfsm);
    buildParseTable(grammar, cfsm);
}

CFSM computeTransitionStates(ConfigurationSet state) {
    CFSM cfsm = empty;
    add state to cfsm;
    state.flag = false;
    do {
        for (each state 'fromState' in CFSM with flag=false) {
            for (each grammar symbol X in grammar) {
                toState = nextState(fromState, X);
                if (toState does not exist in cfsm) {
                    add toState to cfsm;
                    add (fromState, X, toState) to gotoTable
                }
            }
            else {
                propagate lookaheads from 'fromState'
                to 'toState';
                toState.hasPropagatedLookaheads = true;
            }
        }
        fromState.flag = true;
    }
}
```

```
} while (change has been made to cfsm);
return cfsm;
}
CFSM propagateLookaheads(CFSM cfsm) {
    do {
        for (each configurationSet 's' in cfsm) {
            for (each configuration 'c' in s) {
                if (c.hasPropagatedLookaheads) {
                    propagate lookaheads in 'c' to all the
                    configurations
                    in cfsm that have lookaheads propagated from
                    'c' and if lookahead symbols are added set their
                    Respective hasPropagatedLookaheads to true.
                }
            }
        }
    } while (cfsm is changed);
    return cfsm;
}
```

The core of the LALR approach is the generation of the characteristic finite state machine (CFSM) with lookahead propagation, which is then used to drive the parsing process. The LALR parse table generator we have implemented includes 20 classes of about 2,500 lines of code.

4.2. Semantic Analysis

The semantic analysis module deals with implementing the semantics of the user program and translating it to executable code. Semantic processing is performed by many semantic functions, each of which deals with a particular aspect of the semantics. These semantic functions are triggered by what is called the “action symbols” embedded in the BNF. When an action symbol is encountered by the parser, it invokes the corresponding semantic function. Two major data structures, the symbol table and the semantic stack, are used to help the semantic process.

4.2.1. Symbol Table and Semantic Stack. The symbol table is a place where all necessary information about identifiers known to the compiler is stored, such as named constants, variable names, function names, and labels. The most common way of implementing symbol table is using the hash table approach. Since the same identifier may appear in the symbol table at the same time multiple times due to scopes of the user program, the symbol table used in our implementation has some sort of “stack” behavior

In addition to the symbol table that stores information about identifiers, many other types of information are also needed for the semantic processing, such as the address of a branch instruction for an `if`-statement, or information representing a function call. We use a semantic stack to

manage such information. Entries on the semantic stack are called semantic records. Semantic functions communicate with each other through semantic stack. The semantic records at the top of the semantic stack are in-parameters to the next semantic function and upon completion of the semantic function, the “in-parameter” records are popped and new records produced are pushed onto the semantic stack as “out-parameters.”

4.2.2. Semantic Functions. We shall describe three semantic functions (for binary expression, output statement, and function call) as examples of the semantic process and how the code generation process is related to the JVM.

Expressions

We only discuss the expression of the form $E_1 \text{ op } E_2$, where E_1 and E_2 are the right and left operands (may be complex sub-expressions) and op is a binary operator. The algorithm used in this semantic function looks like this:

```
right-operand = SemanticStack.pop();
left-operand = SemanticStack.pop();
Check that the two operands are type-compatible
    with respect to the operator
if (error found)
    semanticError("Incompatible types wrt operator");
Generate code for:
    result = left-operand operator right-operand
Create ExpressionRecord for result with appropriate
information
SemanticStack.push(ExpressionRecord);
```

The code generated depends on the types of E_1 and E_2 and the operator op . Here are some examples:

The code generated for $(x + y)$ where x and y are integer variables would be:

```
iload, x-offset
iload, y-offset
iadd
```

For the Expression with relational operator $(x < y)$, where x and y are integer variables, the code generated would be:

```
iload, x-offset
iload, y-offset
if_icmpge, 00, 07
iconst_1
goto, 00, 04
iconst_0
```

Output Statement

The output statement makes use of the `PrintStream` Class object (in JVM) to display appropriate type variables on the console. The algorithm is shown below.

At the beginning of the output statement,

```
Setup PrintStream instance by entering appropriate
information into constantPool;
Generate code to instantiate static object 'out' of PrintStream
```

When an expression is to be displayed,

```
expr = SemanticStack.pop();
Get the type descriptor of the expression;
Setup print function with valid type descriptor by enterin
appropriate information into constantPool;
Generate code for print(expr);
```

For example, for the output statement $(\text{cout} \ll x ;)$, where x is an integer variable, we will enter the following information into the `constantPool`, where I_i ($i = 1, 2, 3 \dots$) denotes `constantPool` index.

```
I1 - Constant_Utf8, string = 'out'
I2 - Constant_Utf8, string = 'Ljava/io/PrintStream;'
I3 - Constant_NameAndType, name_index = I1,
    descriptor_index = I2
I4 - Constant_Utf8, string = 'java/lang/System'
I5 - Constant_Class, name_index = I4
I6 - Constant_Fieldref, class_index = I5,
    name_type_index = I3
I7 - Constant_Utf8, string = 'print'
I8 - Constant_Utf8, string = '(I)V'
I9 - Constant_NameAndType, name_index = I7,
    descriptor_index = I8
I10 - Constant_Utf8, string = 'Ljava/io/PrintStream'
I11 - Constant_Class, name_index = I10
I12 - Constant_Methodref, class_index = I11,
    name_type_index = I9
```

The code generated would be

```
getstatic, 00, I6
iload, x-offset
invokevirtual, 00, I12
```

Function Call

The algorithm of the semantic function handling function call is given below.

```
Lookup symbol table for 'id', which is the name of the function;
if (not found)
    semanticError("Function not defined");
```

```

Get parameterCount for function 'id' from symbol table;
if (parameterCount != paramCount)
    semanticError("Incorrect number of arguments");
Generate code to invoke the function;
paramCount = 0;

```

As an example, consider the following function call with parameters, and a value is returned.

```

int func(int, float, bool);
x = func(a, b, c);
int x, a;
float b;
bool c;

```

During the function declaration, the following information had been entered in the `constantPool`.

```

I1 : Constant_Utf8, data = 'func'
I2 : Constant_Utf8, data = '(IFZ)I'
I3 : Constant_NameAndType, name_index = I1,
    descriptor_index = I2

```

Now, we enter more information into the `constantPool`:

```

I4 : Constant_Utf8, data = 'Cprog'
    // Cprog is the classname
I5 : Constant_Class, name_index = I4
I6 : Constant_Methodref, class_index = I5,
    name_type_index = I3

```

The code generated is:

```

iload, a-offset
fload, b-offset
iload, c-offset
invokestatic, 00, I6
istore, x-offset

```

Although most of the semantic processing ideas are not new, the main issue of this work is the mapping of these C++ constructs to Java bytecode for the Java Virtual Machine that few have done before.

5. Experimental results

We tested 50 valid user programs with various C++ features and 18 invalid user programs, as shown in Table 1. The numbers of programs tested in the right column are for those programs with the particular C++ features as the main purpose for testing. All of valid programs produced Java bytecode that executed correctly on the Java Virtual Machine. As stated before that the language for our compiler is a subset of C++, only the basic features of C++ are included.

Table 1. Language features tested

language feature	# of programs tested
Variable	26
Constant	14
Assignment	22
Arithmetic expressions	24
Boolean expressions	24
Input & output statements	26
if statement	12
while loop statement	4
for loop statement	6
do-while loop statement	5
switch statement	4
Label, exit, goto	12
void function	12
Value function	8
Parameters	8
Function prototype	10
Syntax and semantic errors	18

Due the page limit, we show the screen dump of only one example below. Figure 1 is the user program echoed during the compilation (the echo usually is turned off) while Java byte code is being generated.

```

cps230.cps.cmich.edu - default - SSH Secure Shell
File Edit View Window Help
Quick Connect Profiles
compiler $ java parse Parse_table dowhile-example.cc FA_table dowhile-example
1. //dowhile-example.cc
2. #include <iostream>
3. using namespace std;
4.
5. int getNum(){
6.     int num;
7.
8.     cin >> num;
9.     return num;
10. }
11.
12. void print(int num){
13.     cout << "sum = " << num << endl << endl;
14. }
15.
16. int main(char args){
17.     int term, sum, N;
18.
19.     cout << "We are to calculate the sum 1+2+...+N." << endl;
20.     cout << "Please enter N: ";
21.     N = getNum();
22.
23.     if (N > 0){
24.         sum = 0;
25.         term = 1;
26.
27.         do {
28.             sum += term;
29.             ++ term;
30.         } while (term <= N);
31.
32.         print(sum);
33.     }else{
34.         cout << "Invalid value for N : " << N << endl << endl;
35.     }
36. }
compiler $ █
Connected to cps230.cps.cmich.edu | SSH2 - aes128-cbc - hmac-md5 - none | 80x38

```

Figure 1. User program

The code is then correctly executed by JVM with valid and invalid inputs, as shown in Figure 2.

```

c compiler $ java dowhile-example
We are to calculate the sum 1+2+...+N.
Please enter N: -10
Invalid value for N : -10

c compiler $ java dowhile-example
We are to calculate the sum 1+2+...+N.
Please enter N: 100
sum = 5050

c compiler $

```

Figure 2. Execution of the bytecode

Although the test programs cover only simple C++ language features, they illustrate that this work has accomplished a substantial portion of a compiler that can be easily extended to include more complex features of the language.

6. Conclusion

In addition to providing cross platform delivery, the Java Virtual Machine provides developers with an opportunity to utilize it not only for the execution of Java programs, but also a wide range of other uses. Languages other than Java can be compiled into bytecode and executed on the Java Virtual Machine in the same way as a Java program. The Java Virtual Machine is gaining popularity among diverse set of applications and with it, the need for methods of utilizing existing programs written in various languages are being considered. This paper demonstrates one possible solution for such problems.

The compiler presented in this paper generates Java bytecode for programs written in C++ with the basic constructs. We implemented an LALR parsing table generator based on the traditional LALR method to build a Characteristic Finite State Machine (CFSM), with modifications to the way the lookaheads in the configuration sets are propagated.

This compiler successfully generates Java bytecode for C++ programs with constructs including variable declaration, constant declaration, assignment statement, input and output statement, if statement, switch statement, for statement, while statement, do-while statement, break statement, continue statement, goto statement, pre- and post-increment/decrement

statements, simple and composite expressions, function declarations, parameter passing, function call, and return statement.

We are working on enhancements to the compiler to include more complex C++ features such as classes.

References

- [1] Frank DeRemer, and Pennello, "Efficient Computation of LALR(1) look-ahead sets", *ACM Transactions on Programming Languages and Systems*, 4(4): pp. 615-649, October 1982.
- [2] Frank Budrus and Jörg Schödel, "Cappuccino – A C++ to Java Translator," *Proceedings of the ACM symposium on Applied Computing*, pp. 660-665, Atlanta, 1998.
- [3] G. Hu and J. Hill, "Binary Level Program Translation to Java Bytecode," *Journal of Electronics and Computer Science*, Vol. 3, No. 1, pp. 25-38, December, 2001.
- [4] Kostas Kontogiannis, Johannes Martin, Kenny Wong, Richard Gregory, Hausi Muller, and John Mylopoulos, "Code Migration Through Transformations: An Experience Report", *Proceedings of CASCON '98*, pp. 1-13, Toronto, 1998.
- [5] Brent Kristensen and Ole Madsen, "Methods for Computing LALR(k) look-ahead", *ACM Transactions on Programming Languages and Systems*, 3(1): pp. 60-82, January 1981.
- [6] Tim Lindholm and Frank Yellin, "The Java Virtual Machine Specification, Second Edition", *Addison Wesley Longman, Inc.*, 1999.
- [7] Andrew Malton, "The Migration Barbell", *First ASERC Workshop on Software Architecture*, 2001.
- [8] Jon Meyer and Troy Drowning, "Java Virtual Machine", *O'Reilly and Associates, Inc.*, 1997.
- [9] Novosoft, "C2J, a C to Java translator", 2001, http://www.novosoft-usa.com/solutions/product_c2j.shtml
- [10] Andrey Terekhov, "Automating Language Conversion: A Case Study", *Proceedings of the International Conference on Software Maintenance (ICSM)*, pp. 654-658, Florence, Italy, November 2001.
- [11] Andrey Terekhov and Chris Verhoef, "The Realities of Language Conversions", *IEEE Software*, 17(6): pp. 111-124, November 2000.
- [12] Eli Tilevich, "Translating C++ to Java", *First German Java Developers' Conference Journal*, Sun Microsystems Press, June 1997.
- [13] C. Cifuentes, B. Lewis and D. Ung, "Walkabout - A Retargetable Dynamic Binary Translation Framework," *Fourth Workshop on Binary Translation*, Charlottesville, Virginia, September 2002.
- [14] Kazuki Yasumatsu and Norihisa Doi, "SPiCE: A System for Translating Smalltalk Programs into a C Environment", *IEEE Transactions on Software Engineering*, 21(11): 902-912, November 1995.