論文
3-8-1

Program Translation from Conventional Programming Source to Java Bytecode
(기존 프로그래밍 원시코드에서 자바 바이트 코드로의 변환)

강 전 근* 김 행 곤**
(Jeon-Geun Kang) (Haeng-Kon Kim)

ABSTRACT

Software reengineering is making various research for solutions against problem of maintain existing systems. Reengineering has a meaning of development of software on exizting systems through the reverse engineering anf forward engineering. Most of the important concepts used in reengineering is composition that is restructuring of the existing objects. Is there a compiler that can compile a program written in a traditional procedural language (like C or Pascal) and generate a Java bytecode, rather than an executable code that runs only on the machine it was compiled (such as an a.out file on a Unix machine)? This type of compiler may be very handy for today's computing environment of heterogeneous networks. In this paper we present a software system that does this job at the binary-to-binary level. It takes the compiled binary code of a procedural language and translates it into Java bytecode. To do this, we first translate into an assembler code called Jasmin [7] that is a human-readable representation of Java bytecode. Then the Jasmin assembler converts it into real Java bytecode. The system is not a compiler because it does not start at the source level. We believe this kind of translator is even more useful than a compiler because most of the executable code that is available for sharing does not come with source programs. Of course, it works only if the format of the executable binary code is known.

This translation process consists of three major stages: (1) analysis stage that identifies the language constructs in the given binary code, (2) initialization stage where variables and objects are located, classified, and initialized, and (3) mapping stage that maps the given binary code into a Jasmin assembler code that is then converted to Java bytecode

요 약

소프트웨어 재공학은 기존 시스템의 유지보수 문제에 대한 해결책으로 많은 연구가 이루어 지고 있다. 재공학은 역공학과 순공학을 이용하여 기존 시스템에 대한 이해와 새로운 시스템의 개발을 의미하며 기존 시스템에서의 컴퍼넌트들로부터 필요한 기능을 가져와 재구성 하는 것이다. 본 논문에서는 기존의 프로시져 언어에 의해 컴파일된 바이너리 코드를 입력으로 받아서 웹 기반 자바 바이트 코드로 변환한다. 즉 바이너리-바이너리 단계에서 수행되는 소프트웨어 시스템을 제안한다. 이를 위해 먼저 Pascal-L 에 의해 작성된 기존의 프로그램 언어를 Jasmin 이라는 어셈블리 코드로 먼저 번역하고 사용자 읽기 가능한 자바 바이트 코드 상태인 Jasmin 어셈블리가 실제 자바 코드로 변환된다.

* 정회원 : 대구 영진전문대학 컴퓨터계열부 교수
** 정회원 : 대구 가톨릭대학교 컴퓨터정보통신공학부 교수

논문접수 : 2002. 6. 22.
심사완료 : 2002. 7. 20.

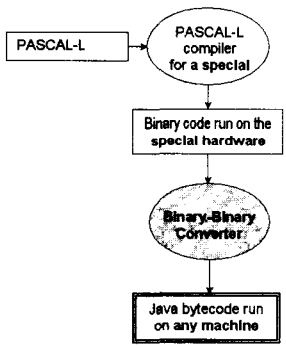
이 시스템은 결국 기존의 원시코드가 번역기를 통해 실행 가능한 바이너리 코드 형식으로 실행된다. 이 번역과정은 먼저 주어진 바이너리코드에서 언어구조를 식별하는 과정과 변수 객체의 위치를 분석하고 초기화하는 과정 그리고 주어진 바이너리 코드를 Jasmin 코드로의 매핑하는 단계등으로 구성된다.

1. Introduction

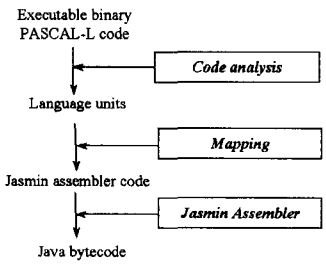
For today's enterprises, information integration is one of the top priorities for business success. There are many aspects in dealing with information integration issues. For example, uniform object modeling at a higher level, multi-tier architecture of software components at the middle levels, and reusable library APIs at a lower level, have all been designed and developed for the purpose of integrating information of different formats on heterogeneous platforms. However, few have considered information integration at the lowest level ¾ executable binary code integration [3, 9]. Much of the executable code existing within organizations today is compiled from source programs written in diverse programming languages. It would be very desirable if these binary codes could be converted into a platform neutral format so that they could be executed on different machines. If this could be done, a natural choice of a target format is Java bytecode [6]. Java was born with the "write once, execute everywhere" philosophy that typifies platform independency [2, 10].

In this paper, we present a prototype of such a translation system to translate the compiled binary code of a procedural language into Java bytecode. This Pascal-like procedural language, called PASCAL-L [4] that we designed and developed ourselves, includes the basic structures found in many languages, including local and non-local variables, constants, expressions, control statements, input and output, and subprograms. The PASCAL-L compiler, just like any other compiler, compiles a PASCAL-L program and generates binary code for execution on the PASCAL-L "hardware" (i.e. a virtual machine).

The executable binary code is based on the traditional register-memory architecture. That is, most instructions of the executable code involve either registers or memory addresses, or both. Because the format of the PASCAL-L binary code is known (will be explained later), we will be able to decode the instructions, analyze them, and map them to Java bytecode. To do this binary level conversion, the process bridges the gap between traditional compilers and Java bytecode, as shown in [Fig. 1] where our work is the circle in shaded pattern.



[Fig. 1] Binary level code translation



[Fig. 2] The overall translation process

The translation process is done in the three modules. First, the binary code of the compiled PASCAL-L program is analyzed to find program units. Then these program units are mapped to

Jasmin assembler code according to some matching patterns of the two. Finally, the Jasmin code is converted to Java bytecode. The process is shown in [Fig. 2]

Our work is focused on the first two modules: code analysis and mapping. The Jasmin assembler, developed by Jon Meyer and Troy Downing [7], is publicly available software, which assembles Jasmin assembly code into Java bytecode.

The code analysis module is the key part of the process. It identifies various units in the program code by discovering patterns of instructions.

The mapping module maps the program units discovered into corresponding assembly code in Jasmin, which is at a low level, with an almost one-to-one correspondence with Java bytecode.

In this paper, we shall first briefly describe in section 2 the specifications of the three language forms, namely, the PASCAL-L binary code, Jasmin assembly code, and Java bytecode. The code analysis module is given in section 3 and the mapping module is in section 4. We shall then give some examples to illustrate the translation process in section 5. Finally, we conclude the paper in section 6 with a discussion of our findings.

2. Language Specifications

In order to translate the PASCAL-L binary code to Java bytecode with Jasmin assembly code as an intermediate bridging format, we will first need to introduce the format of each of the three codes. They are low-level language specifications.

2.1 Java Virtual Machine

Java virtual machine (JVM) [6] was designed with portability in mind. To achieve platform independence, Java compilers generate Java bytecode, which is designed to run on any platform as long as a JVM is installed on the machine. Java bytecode is platform neutral and interpreted by the JVM. Programs written in languages other than Java can also be compiled into Java bytecode and executed on the JVM, if such compilers exist. Also, one can directly program the JVM [1, 5, 8].

An implementation of a Java virtual machine is called a Java runtime system, and it typically consists of the following basic components: an execution engine, a memory manager that manages the heap and garbage collection, an error and exception manager, support for native method libraries, a threads interface, a class loader, and a security manager. A runtime data area is also kept which normally contains space for the program counter, Java virtual machine stacks, the heap, a method area, a runtime constant pool, and native method stacks.

The Java virtual machine functions by loading correctly formatted Java class files and executing the bytecode they contain. The Java instruction set has 160 instructions (can be extended to 256) for various operations including method invocation. Java data types include primitive types (byte, int, float, char, double, etc.) and reference types (class, interface, array, etc.). Each instruction may include operands that are generally pushed on an operand stack and then operated on by the opcode. For example, the sequence of instructions

```
iload 4     // push int variable 4 on stack
iconst_3"   // push constant 3 on stack
iadd        // pop 2, add, push result on stack
i2f         // convert stack top from int to float
fstore 5// pop stack top and store to float var 5
implements float var5 = (float) intVar4 + const3.
```

An L followed by the full class path denotes class types. For example, **Ljava/lang/String;** denotes the String class in java.lang. Array types use the '[' character followed by the type descriptor of the type of the array elements Here are some examples:

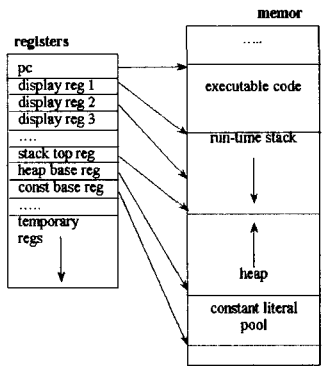
[D 1-D array of doubles
[[I 2-D array of integers
[[Ljava/lang/String 3-D array of Strings.

Type descriptors of methods are of the format (<argument_types>)<return_type>. For example, (SF[Ljava/lang/Thread;)I represents a method which takes arguments of type short, float, and Thread[]; and returns an integer.

Another aspect of the JVM that is important to our work is the class loader. It checks, among other things, the internal structure of the class file to verify the integrity of the class and the bytecodes it contains

2.2 Architecture and Instruction Set of PASCAL-L

The PASCAL-L system is a virtual machine that includes hardware architecture and an instruction set. The similarity of the PASCAL-L virtual machine and the Java Virtual Machine is that they share the "virtual machine" in their names. But there is very little else in common between the two. PASCAL-L is a traditional machine with a set of 64 registers and a memory of N words. The partition of the user space in memory consists of executable instructions, a run-time stack, a literal pool where constants are stored, and a heap for dynamically allocated data, as shown in [Fig. 3].



[Fig. 3] PASCAL-L hardware architecture

We can see in [Fig. 3] that the registers are further divided into three groups: a fixed number of display registers that are used as base registers of variable areas on the run-time stack, a fixed number of reserved registers for special purposes (such as for program counter pc, the base and top of the run-time stack, the base of the heap, etc.), and the remaining as temporary registers used for any other purposes, including to hold intermediate results of expressions.

The PASCAL-L's instruction set includes instructions for the five built-in types: integer, real, boolean, char, and string. [Fig. 4] shows the format of the 32-bit instructions.

opcode	register	base	offset
6 bits	6 bits	6 bits	14 bits

[Fig. 4] PASCAL-L instruction format

The 6-bit opcode field may be any of the possible 64 instructions, such as arithmetic, branch, memory-register data transfer, and other miscellaneous instructions. If an instruction references a memory location, the (base, offset) pair often represents the effective memory address, where base is the base register containing the starting address of a memory area and offset is the displacement within the area. This addressing mode is particularly useful for accessing values in various parts of the memory. Instructions applied to data of different types have different opcodes. For example, ADDI and ADDR are arithmetic addition operations for integer and real values, respectively. This information is necessary for us when we try to identify variables and their types from the binary code.

2.3 Jasmin Assembly Code

Jasmin is a freely available Java class file assembler that takes ASCII descriptions of Java classes written in a simple assembler-like syntax

using the JVM instruction set. It converts them into binary Java class files.

Jasmin programs consist of three types of statements: Jasmin-specific directives, instructions for the Java virtual machine, and labels. Since most of the Jasmin assembly instructions are pretty much the same as the Java bytecode (in human-readable format), we only describe here the directives and Jasmin file structure. Jasmin directives (such as **.class, .method, .field, .var,, etc.**) may take parameters, very much like bytecode instructions. Class names are written using the full path with '/' as the separator. Methods are represented using the full path of the method name followed by the descriptor as discussed before. For example, the println method of the **PrintStream** class is represented as

```
java/io/PrintStream/println(Ljava/lang/String;)V.
```

Field names are specified using two tokens. The first provides the class and name of the field and the second is its descriptor. For example, the instruction

```
getstatic java/lang/System/out Ljava/io/
OutputStream
```

gets the static value of the object out of the class **java.lang.System**, with type **java.io.Output-Stream.**

A Jasmin file consists of the following parts: a header, a list of field definitions, and a list of method definitions. The header includes specific information on the class being created, such as the class file name, class name with access specifier (**public, final, super, interface, or abstract**), and the parent class. If the class is not extended from another class, then the parent is the Object class, **java/lang/Object.**

Following the header information is a list of field definitions, if any. The **.field** directive is used and the format is:

```
.field <access-spec> <field-name>
       <descriptor> [ = <value>]
```

Here **<access spec>** is zero or more of the keywords: **public, private, protected, static, final,** volatile, and transient. The **<field-name>** indicates the name of the field and **<descriptor>** refers to its type descriptor. The field may be initialized with a constant value.

The remainder of a Jasmin file contains of a list of method definitions. The basic format of all methods is:

```
.method <access-spec> <method-spec>
<statements>
.end method
```

The access specifiers are zero or more of: **public, private, protected, static, final, synchronized,** native, and abstract. The **<method-spec>** is the name of the method and its type descriptor. <Statements> refers to the code contained in the method.

3. Code Analysis

In order to achieve the translation of binary code into Jasmin assembler code, it is necessary to analyze each line of the binary code and discover higher-level constructs. This is very much like partial de-compilation — discovering program structures from the binary code and re-constructing them into a high-level representation.

The analysis stage involves rebuilding as complete a picture as possible of the original structure of the program. The main program must be distinguished from any of its nested procedures, all of which will be mapped into Java methods. A complete symbol table must be reconstructed that

will represent all variables, the methods in which they first appear, their scopes, and their types. All literals, their types and their actual values must be retrieved. Finally, information must be collected about the branching structure of the program in order to prepare for converting from machine code branching instructions to the higher-level construct of branch statements in combination with labels.

Because the compiled binary code of a PASCAL-L program uses registers to point to the beginning of areas in the memory, as shown in [Fig. 2], it is necessary to pass the knowledge of which register is used for which memory area to the code analysis stage. This knowledge is used to infer the category of information being accessed. For example, if we know register 14 is the base register of the constant literal pool, we will treat it differently from the case if register 14 were the base of the temporary pool, which would hold an intermediate result of an expression being evaluated rather than a constant literal.

Once the preliminary information becomes available, we can start to find and identify the program structures in the binary code.

3.1 Discovery of Subprograms

PASCAL-L, like many other procedural languages, allows nested scopes with the traditional scope rules applied. Variables in the main program (global scope) will be mapped to class fields in Java. Non-local (but not global) variables are also mapped to class fields even some outer scopes may not have access to them. This won't create problems because the PASCAL-L compiler would have discovered the access violation if there were any such violations.

In examining machine code in order to map the PASCAL-L subprograms into Java methods, it's necessary to extract four crucial pieces of information. First, the start and finish points of the procedure in the machine code must be identified. These are located by matching patterns of sequences of the opcodes. Second, the procedures must be categorized in a way that makes them distinguishable from one another since they no longer retain the identifiers used to represent them before compilation. Third, the instructions the procedures contain must be marked as belonging to that method. Finally, the procedure's relationship with the other procedures must be determined, so that a picture of the original structure of the program can be reconstructed.

One pass through the code is sufficient to meet all of the above requirements. The machine code is traversed sequentially with the opcodes inspected until a pattern for entry into a procedure is located. Because the procedure-entry code of all PASCAL-L procedures share the same pattern that is before the code of the first executable statement within the procedure, we can identify them by matching the pre-defined pattern. This procedure-entry code is for pushing an activation record on the run-time stack, save and set the display register, update the stack top, etc. Similarly, we can match the procedure-exit pattern when it returns to the caller. Some of the patterns look like:

Pattern Name	Code Pattern
StartMain	LDA, ADDI, LDA, ST
Proc	LDA, ADDI, ST, LD, ST, ST
ProcReturn	LDA, LD, LD, B
ProcCall	LDA, ADDI, LD, LDA, ST, B
ProgramEnd	HALT

For subprograms with parameters, the procedure-entry code pattern remains the same. The code for parameter passing is at the place of the call rather than as part of the procedure-entry code. There are some differences in calls with or without parameters, but these differences (LD and ST instructions for copying value parameters and/or addresses of reference parameters) appear before the code of **ProcCall** and therefore do not affect the way the subprogram's starting and ending are matched.

Once the entry point of a subprogram is identified, a new method object can be entered into the method table, with a reference to its starting location. Each method object will eventually contain all of the information required to generate the Jasmin code. Each instruction contained within the procedure is then marked as belonging to that method up until the exiting pattern of the procedure is discovered. At that point, a reference to the ending location can be entered into the method table. Because the PASCAL-L code does not retain identifiers of the subprograms, we simply assign a unique number to each method. The methods are then identified as methodn in the Jasmin code.

Since PASCAL-L is a Pascal-like language, in which the subprograms are in the declaration portion of an outer subprogram, the code of the outer scope won't be generated until the code of all the nested subprograms declared within it are generated. To establish the nesting/enclosing relationships between subprograms, we start with the scope 1. It is incremented by 1 when a procedure-entry pattern is recognized and decremented by 1 when a procedure-exiting pattern is identified. There may be multiple subprograms with the same scope (say scope n) in a sequence. Once the end of the sequence is reached, we know that these subprograms are nested in the next subprogram of scope n - 1. This pattern may repeat until the main program is reached that has scope 1.

3.2 Discovery and Categorization of Variables

After subprograms are found and matched to Java methods, the next step is to locate the variables that the JVM will refer to in the bytecode. The variables must be typed so that the appropriate typed instructions can be used when they are referenced.

3.2.1. Locating and Identifying Variables

Variables can no longer be referenced by their names because the names no longer exist in the machine code. A variable is recognizable in the machine code only as a reference to a (base, offset) pair of values. The base indicates the scope of the variable and the offset is the memory offset of the variable in the activation record on the runtime stack.

Unfortunately the (base, offset) pair is not unique for every variable because multiple procedures may have the same scope, and multiple variables within these procedures could all be identified using the same offsets.

Discriminating the procedure in which a particular variable was declared is not as simple as locating the first procedure in which that variable is referenced. For instance, a variable may be declared in the main program, but is not referenced within the main program but within a nested procedure instead. Thus, the variable has a scope 1 (that of the main), but is used only in a procedure of differing scope. This means that the first indication of which procedure a variable belongs to is found by comparing the scope of the variable with successive outer methods until a match is found.

There are circumstances where many procedures exist with a scope matching that of the variable, and the variable was never referenced in its declaring procedure. In this situation, we need to reexamine the nesting hierarchy so that it's possible to determine which of the procedures with matching scope is an outer procedure, either directly or indirectly, of the procedure in which the variable was referenced. An illustration is given in the following PASCAL-L program.

```
program testScope;      { main program, scope 1 }
  procedure A( );       { procedure A, scope 2 }
    var y : integer;
    procedure B( );     { procedure B, scope 3 }
    begin
      y := 5;           { variable y, scope 2 }
```

```
    end;
  begin
    B( );
  end;
  procedure P( );       { procedure P, scope 2 }
    var x : integer;
    procedure Q( );     { procedure Q, scope 3 }
    begin
        x := 2;         { variable x, scope 2 }
    end;
  begin
    Q( );
  end;
begin                   { main starts here, scope 1 }
  ...
end.
```

The two procedures, A and P, both of scope 2, each contain one nested procedure of scope 3, *B* and *Q*. A and *P* each contain a variable declaration; however, the variables are not accessed within *A* or *P*, but instead within *B* and *Q*. At first glance, it's easy to see that the variable *y* belongs to procedure *A* and the variable *x* belongs to procedure *P*. This is not as clear when the machine code is examined. The code pattern of both variable references would be similar to the following:

```
procedure-entry code
LD   R constant
ST   R 2 0     { var in scope 2, offset 0 }
procedure-exit code
```

To solve this problem of same (base, offset) of variables, we trace the subprogram nesting hierarchy to find instructions that reference the variables. Because we know the methods these instructions belong to, we will be able to identify the variable's method.

Once a variable has been identified, a key is formed for it and entered into a symbol table. The key is made up of a combination of the variable's base, offset, and parent method number. The variable is given a numeric identifier, which the JVM will use to reference it. The variables are numbered on a per method basis, starting at 1, since 0 is reserved for a reference to the class instance, *this*. If method calls with parameters were to be created, the parameters would be numbered first after the reference to *this*, before the local variables.

3.2.2 Identifying Types of Variables

Every variable must be typed as required by the JVM. Since all variable declarations in the PASCAL-L program do not generate code, only the references to the variables in the executable statements provide type information in the machine code. Some of the PASCAL-L opcodes are typed, such as ADDI and ADDR for integer and real addition operation, respectively. We can use these typed instructions to figure out the types of their operands, and to extend these to data items in other non-typed instructions. For example, if the code for x - y is

```
LD      R   Basex   Offset_x
SUBI    R   Basey   Offset_y
```

we will know that the type of y is integer and infer that the type of x is also integer. The type would be float if the opcode of the subtract instruction were SUBR. If one of the two variables is integer and the other is float, the integer variable would have been converted to float first using the FLOAT instruction. Furthermore, we may be able to infer the type of the operand of a non-typed ST (for STORE) instruction if it is preceded by a non-typed LD (for LOAD) instruction with an operand of a known type. That is, if we have the code sequence

```
LD      R   Basex   Offset_x
ST      R   Basey   Offset_y
```

and we know that the data item x is of type T, then we can infer that the type of data item y is also of type T. It is worthwhile to mention here that the **load** and **store** instructions in Java bytecode are typed to satisfy the strong typing

requirement of JVM. PASCAL-L's LD and ST instructions are type-less because they treat the data items as a 32-bit bit pattern in a register or a memory location, regardless of the interpretation of the bit pattern.

Many other PASCAL-L instructions also provide type information, such as READ and WRITE (RDI, RDR, RDCH, WRI, WRR, etc.), as well as branch instructions (BZI, BZR, etc.).

Of course, discovering types of variables may not succeed if an opcode of an instruction cannot be found that applies to the variable. In the above LD-ST example, if we do not know the type of x, we won't be able to infer the type of y. If x = y is the only statement in the program (which generates the LD and ST instructions) referring to x and y, there is no way we can recover their types. It turns out, though, this is not a problem at all, we simply discard the code without determining their types and no Jasmin code will be generated for them. Omission of the code won't have any adverse effect on the results because x and y are never used in the program. This exclusion can be considered an optimization rather than a failure to determine type.

3.3 Discovery of Constants

In addition to variables, we must also find and categorize constant literals. They are provided in the literal pool of the binary code and can be located by their references in the instructions. These references always use the literal pool register as the base register, so it is straightforward to locate these references. Once a constant literal is located, we use the same approach as we did for variable types to associate the constant with an appropriate type. All constant literals are of primitive types, e.g. integer, real, character, boolean, and string. Except for strings, the other four types have fixed lengths so that retrieving a value is just a matter of getting the contents of the code at the particular address plus the length.

Since the values are stored in integer format, integer-type values are as the way they are stored. For real-type items, their internal representation (bit pattern written as an integer) is reinterpreted as a real number. This can be done using a structure like a union in the C or C++ language. In Java, the method intBitsToFloat(I)F in the package java.lang.Float provides a similar functionality.

Character and boolean types are both stored in the Java virtual machine as integers, and so no effort is required to convert them. Strings, on the other hand, are stored as the string characters preceded by the length. The length is stored in a single 32-bit word at the index that is used to reference the string. Therefore, to get the value of a string constant, we will read the length first, and then extract the characters from the following words to reconstruct the original string.

3.4 Locating Branches and Labels

The final step before the actual conversion of the PASCAL-L program to Java is to find all branches in the machine code and their target addresses that will be labels in Jasmin. Branches (or jumps) represent all control statements in PASCAL-L, including selection, repetition, exit, continue, return, etc. These branch instructions are identified in their opcode, such as:

B	unconditional jump to the target address
BZ	branch if (R) = 0
BGZI	branch if (R) > 0, integer
BGZR	branch if (R) > 0, real
etc.	

where R is the register used in the instruction. The target address is specified in the branch instruction as well, but most likely in the form of (base, offset) rather than the direct physical address. These target addresses will be mapped to labels in Jasmin.

A branch may jump forward or backward to the destination. If it were a backward jump, the target

destination would be processed before the branch instruction was encountered. When we reach the branch instruction, a new label is created and assigned to the destination that is known. If it is a forward jump, the destination is unknown at this time. So we record the address of the branch instruction in a "label-to-be-resolved" list. When the jump target is reached, we create a new label for the destination and backtrack it to the jump instruction found in the list.

3.5 Construction of Input and Output Objects

One more issue remains before we can proceed with code conversion — creating I/O objects. The Java language was designed to be secure and platform independent. This independence means that instructions that are platform dependent, such as input and output, must be abstracted. For this reason, there are no instructions in the Java virtual machine instruction set for direct reading and writing of data. Instead, all reading and writing must be accomplished using the **java.io** and **java.lang** packages. In contrast, the PASCAL-L language provides opcodes for directly reading and writing all of its primitive types: integer, real, boolean, character, and character string.

In order to read and write utilizing the Java packages, it's necessary to create instantiations of the required objects and reference the reading and writing methods if the program actually involves read and write. These decision can be made by inspecting the opcodes to see if instructions like RDI (read integer), WSTR (write string), etc. exist in the code. We create the I/O objects on a per method basis when a read and/or write opcode is found during the procedure-mapping phase.

There are a lot of details about creation and use of I/O objects and their methods, such as format read and write. Since they are not very critical to the principle methodology of our code translation strategy, we will not discuss these details any further.

4. Mapping PASCAL-L Code to Jasmin Syntax

The next stage is to actually generate Jasmin code. This is accomplished by pattern recognition of all program constructs in the PASCAL-L code and mapping the patterns to their Jasmin equivalents. The program constructs fall into the following five categories:

- Main program, subprograms and calls
- Simple constructs
- Input and output operations
- Expressions
- Control constructs

Before we map each of the program units to Jasmin, we first need to create a class to represent the PASCAL-L program. This step involves generating a class definition, the **<init>()** method that is the instance initialization method for the class, and declaring any class fields. A class that will be used to represent a PASCAL-L program (or any other procedural language) will not require inheritance and can simply follow the default rule and extend the **java.lang.Object** class. Declaring the fields is simply a matter of listing all of the fields found in the symbol table with their initialization values. The basic Jasmin syntax to depict classes that represent PASCAL-L programs is given below:

```
.class public classname
.super java/lang/Object
;insert field list here, such as:
.field public static field3 I = 0
;instance initialization method
.method public <init>( )V
   aload_0
   invokespecial java/lang/Object/<init>( )V
   return
.end method
; insert main method here
; insert any additional methods
```

Pattern Name	Opcode Pattern
Subprogram Patterns:	
StartMain	LDA, ADDI, LDA, ST
Proc	LDA, ADDI, ST, LD, ST, ST
ProcCall	LDA, ADDI, LD, LDA, ST, B
ProgramEnd	HALT
Simple Construct Patterns:	
Load	LD
Store	ST
Assign	LD, ST
Input and Output Patterns:	
Write	Pattern 1: LDA, WI-WSTR
	Pattern 2: WLN
Read	Pattern 1: LDA, RDI-RSTR
	Pattern 2: RDLN
FormatDecimal	LD into width register
MakeWidth	LD into width register + 1
Expression Patterns:	
EvalExpr	Pattern 1: LD, ADDI-DIVR, ST
	Pattern 2: ADDI-DIVR, ST
	Pattern 3: ADDI-DIVR
TypeConversion	Pattern 1: LD, INT or FLOAT
	Pattern 2: LD, INT or FLOAT, ST
EvalUnary	Pattern 1: NEGI or NEGR
	Pattern 2: NEGI or NEGR, ST
EvalUnaryNot	LD, BZ, ST, B, ST
EvalBoolWithoutLoad	BZ-BNLZR, ST, B, ST
EvalBoolWithLoad	LD, BZ-BNLZR, ST, B, ST
EvalBoolAnd	Pattern 1: BZ, LD, BZ, B, ST, B, ST
	Pattern 2: LD, BZ, LD, BZ, B, ST, B, ST
EvalBoolOr	Pattern 1: BNZ, LD, BNZ, ST, B, ST
	Pattern 2: LD, BNZ, LD, BNZ, ST, B, ST
Control Statement Patterns:	
If	LD, BZ
For-Loop	LD, SUBI, BGZI or BLZI
SimpleLoop	B

[Fig. 5] Opcode patterns for program constructs

Pattern matching identifies the program units. The patterns for each of the five categories are listed in [Fig. 5].

In examining the patterns, it becomes apparent that many are very similar, with the only difference being a leading load, LD, or a tailing store, ST. This is due to the very nature of the machine code created by the PASCAL-L compiler. In many cases, values may be left in a register at the end of an operation and no leading load instruction is necessary before the next operation commences. Similarly, the lack of a store instruction at the end of a pattern indicates a case where the result was left in the register rather than stored.

The basic process of mapping these opcode patterns into Jasmin syntax involves abstracting their function into a higher-level construct wherever possible. This means that references to registers and memory locations must be translated into references to local variables, fields, and literals. Branch opcodes will become Java branching instructions referencing labels placed in the code. Sets of opcodes will be transformed into higher-level selection and repetition constructs or methods. This disassembly stage completes when all opcodes have been mapped to Jasmin syntax.

4.1 Mapping Subprograms

The PASCAL-L main program maps into the Java method **public static main([Ljava.lang. String;)V** and a subprogram maps into anonymous methods as methodn, where n is the index of that method in the methods table, similar to the naming convention used for class fields. For instance, a method at index 3 will have the signature **public static method3()V.** In addition, the number of local variables and maximum operand stack size are calculated based on the information in the symbol table. The bytecode verifier of the JVM will use these values. The shell of a method in Jasmin is like this:

```
.method public static method3( )V
.limit stack 2
.limit locals 5
; insert input/output object creation here
; insert local variable initialization here
; insert body statements here
return
.end method
```

The statements in the body of the method are then mapped to corresponding Jasmin code, as discussed in the following sections.

During processing of body statements of methods, patterns for procedures calls may be encountered, consisting of the opcode pattern LDA, ADDI, LD, LDA, ST, and B, which represent the instructions necessary to push the procedure's activation record on the runtime stack, set it's return address, and jump to the procedure. This sequence of events requires the generation of a single line of Java assembly code to invoke the method and accomplish the same effect. Again, the manipulation of registers and the runtime stack is not seen at this level of abstraction. The **invokestatic** instruction is used to invoke static methods, and so the syntax for invoking the method from the previous example would be **invokestatic classname/method3()V.**

4.2 Mapping Simple Constructs

Simple program constructs are load and store instructions that move data from memory to register, or from register to memory. They occur very often in the PASCAL-L binary code, particularly for expression evaluations and assignment statements. In Java bytecode, however, the load and store instructions are applied to the operand stack and variables. The types of the variables and constants we extracted from the PASCAL-L code, as we discussed in section 3.2.2 and 3.3, are used to generate Jasmin (and hence JVM) instructions that are typed (such as iload for loading integer, dload for loading double, etc.). For example, the assignment statement **x := y - 3.5** would be compiled to PASCAL-L code as the follows, assuming x and y are of type real:

LD	R	$Base_y$	$Offset_y$
SUBR	R	$Base_{literal-pool}$	$Offset_{3.5}$
ST	R	$Base_x$	$Offset_x$

The corresponding Jasmin and Java code would be

```
fload   4
ldc     3.5 // load constant from constant pool
fsub
fstore  3
```

where 3 and 4 are variable ids for x and y.

One issue that needs special consideration is the handling of temporaries in the PASCAL-L code. They are registers holding intermediate values during the evaluation process of expressions. Since a temporary may hold an integer value at one time and a float value at another during execution, we cannot attach a fixed type with the temporary. One solution would be to simply keep the temporary on the operand stack just like any other data item until it is used. We need to keep track of the type of the temporary when it is used in different cases and maintain the accurate count of the stack size, so that the Java bytecode verifier will not raise an exception. Another way to handle this is to create a local variable for the temporary and push that variable on the stack. That variable will be treated as other variables so that the type of the variable and the maximum stack size will be fixed. To accomplish this, we can use arrays, one array per type, to hold the variable ids. When the temporary is encountered, we go to the array according to the temporary's type and assign an id of a variable to be used in place of the temporary.

4.3 Mapping Input and Output Instructions

In the JVM, the actual physical implementation of the host machine and its input and output capabilities are abstracted. As a security measure and to support cross-platform operability, the JVM does not allow operations that directly read and write to standard input and standard output. These must instead be performed using the facilities provided by the java.io package. Due to the much

higher level of abstraction, input and output statements will require a significant effort to achieve an equivalent mapping. A PASCAL-L write statement is of the form WRITE(<output-list>) that is compiled into a sequence of (LDA, WR) instruction pairs, each of which writes one value in the <output-list>. For example, the WRITE("x = ", x) will generate the following code:

LDA	R	$Base_{literal-pool}$	$offset_{"x = "}$
WSTR	0	R	0
LDA	R	$Base_x$	$Offset_x$
WI	0	R	0

In contrast, to output the same list of values in Java will involve the following steps:

Construct and push on the operand stack a **java.io.PrintStream** object that will be used for the actual writing of the values.

1) Construct a **java.lang.StringBuffer** object that will hold the string representation of all the items in the **<output-list>**.
2) Each value in the **<out-list>** is converted into a **java.lang.String** object and added to the **StringBuffer** object.
3) Once all the values have been added to **StringBuffer,** its **toString()** method is invoked and the complete string is left on the operand stack top.
4) Finally the **PrintStream** object is responsible for writing the string.

Rather than creating the **PrintStream, StringBuffer,** and **String** objects every time an output statement is mapped, we create these objects on a per method basis at the procedure-method mapping time that are treated as local variables.

Another complication is the formatted output that involves the width specification for the value to be written and the width of decimals for real numbers. We will not discuss the details of handling this issue because it is not very important to this paper.

4.4 Mapping Expressions

PASCAL-L code of arithmetic and boolean expressions contain specific patterns that can be mapped. Because an expression can be defined recursively like this: a primitive item (constant, field, local variable, etc.) is an expression; if **e1** and **e2** are expressions, **e1 op e2** is also an expression where op is a binary operator, so is **(e1)**. Hence, we will not need to consider the complicated inner structures of **e1** and **e2**; rather, they are just treated as simple values. An example of converting such an expression was given in section 4.1. Here we omitted unary operators that are quite trivial to handle anyway.

Boolean expressions are treated quite differently with a sequence of opcode resulting in 1 or 0 (true or false). The PASCAL-L code for **j < k**, for example, would look like:

LD	R	$Base_j$	$Offset_j$
SUBI	R	$Base_k$	$Offset_k$
BLZI	R	0	pc + 3
ST	ZeroReg	$Base_{temp}$	$Offset_{temp}$
B	0	0	pc + 2
ST	OneReg	$Base_{temp}$	$Offset_{temp}$

This code will leave the result (1 or 0) in a temporary register. Similarly, other comparison operators will have an identical code pattern except that BLZI is replaced by another appropriate opcode, such as BZ, BGZI, etc. For each of these code patterns, we can convert it to Jasmin according to the mappings shown in [Fig. 6].

Using this mapping, the above PASCAL-L code for j < k will be converted to the following Jasmin code:

```
; assume j is variable number 5
; assume k is variable number 6
iload 5
iload 6
isub            ; arithmetic expression  j - k
iflt label1     ; go to label 1 if less than 0
iconst_0
istore 7        ; store 0 for false
```

```
goto label2     ; expression done
label1:
iconst_1
istore 7        ; store 1 for true
label2:
```

PASCAL-L opcode	Java Bytecode
BZ	ifeq label
BZ	fconst_0 ifcmpl ifeq label
BNZ	ifne label
BNZ	fconst_0 ifcmpl ifne label
BGZI	ifgt label
BGZR	fconst_0 ifcmpl ifgt label
BNGZI	ifle label
BNGZR	fconst_0 ifcmpl ifle label
BLZI	iflt label
BLZR	fconst_0 ifcmpl iflt label
BNLZI	ifge label
BNLZR	fconst_0 ifcmpl ifge label

[Fig. 6] Branch Correspondence of PASCAL-L and Java

4.5 Mapping Control Statement Patterns

Once the boolean expressions are stripped out from the PASCAL-L code, the remaining code for control statements are just a bunch of branch instructions. Here we just show the code patterns of the if-statement and the for-loop-statement, in both PASCAL-L and in Java.

The if-statement in PASCAL-L is pretty much similar to that in many procedural languages. It has the optional else-part and each of the then-part and the else-part may be any statement including a nested if statement. The complied code would look like this:

```
Code for eval bool_expr1, result in a temp
LD      R       Base_temp       Offset_temp
BZ      R       0               Eval bool_expr2
Code for statement_list1
B       0       0               jump_out_location
Code for eval  bool_expr2, result in a temp
LD      R       Base_temp       Offset_temp
BZ      R       0               Eval bool_expr3
Code for statement_list2
B       0       0               jump_out_location
.......
Code for statement_list(n+1)
jump_out_location:
```

It is clear from this example that the control statements would only concern the values of the boolean expressions involved in the control statements with some branch instructions to accomplish the intended control. The Jasmin code for the same structure is shown below.

```
; insert boolean expression 1 here
label1:
iload 0
ifeq label2
   ; insert statement list 1 here
goto label3   ; jump out
label2:
   ; insert boolean expression 2 here
label5:
iload 0
ifeq label6
; insert statement list 2 here
goto label3   ; jump out
label6:
; insert statement list 3 here
.....
label3:     ; out location
```

A for-loop statement involves a control variable with an initial value and a boolean expression as the loop termination condition. The PASCAL-L code pattern for its for-loop statement is *LD*, *SUBI*, *BGZI* or *BLZI*, which is recognized during the analysis stage. This translates to a simplified binary arithmetic expression with no storing of the result followed by

one of two possible branching statements, iflt or ifgt. The tail of the for-loop would contain an increment or decrement of the control variable, followed by a single branch statement to route flow back to the beginning of the loop. The Jasmin code would be similar the following:

```
label4:
iload 7
iload 8
isub
iflt (or ifgt) label5
   ; for-loop body statements here
   ; inc or dec loop counter, say, by 1
iload6
iconst_1
iadd
   ; branch back to the beginning of loop
goto label4
label5:
```

Other control statements like while-loop and switch/case-statements are handled in a similar way.

5. Assembly Jasmin to Java Bytecode

Once all of the PASCAL-L instructions have been mapped and the complete Jasmin code has been generated for all of the methods, the final stage of assembly can commence. Methods where an exception was thrown must have a pop statement added to the end. Then all methods must have a return statement added to indicate to the Java virtual machine that it should return from the method call. An **.end** Jasmin directive is then added to indicate end of the method to the Jasmin assembler. Finally, all sections of code thus far generated must be assembled to form a complete Java class.

The class name provided by the user is utilized in the class declaration. In addition to declaring the class, code to represent the **<init>** method of the class must be generated. There is no concept of a constructor method in the PASCAL-L, so no additional constructor methods must be generated. The **<init>** method will remain constant for all classes. Field declarations and initializations must be assembled and added to the class. Finally, all methods will be inserted into the class, starting with the main method. The final Jasmin code created will appear like the one shown in the beginning of section 4.

The complete Jasmin code is saved in a file with suffix .j required by the Jasmin assembler that generates the corresponding Java bytecode. We want to emphasize here that the Jasmin assembly code is very much like Java bytecode. In fact, all instructions in Jasmin assembly code are identical to Java bytecode. The main job of the Jasmin assembler is in converting the human-readable Java bytecode to binary Java bytecode for execution by a JVM.

6. Experimental Results

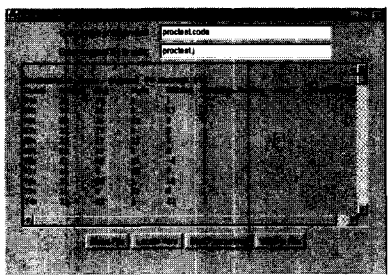
We tested a wide range of PASCAL-L programs that include various language structures like arithmetic and boolean expressions, assignments, control statements, read and write with format, and subprograms. Because the purpose of our testing was to make sure each step of the translation is done in the correct way, we developed a graphical user-interface to allow the user to go through the stages one at a time, including reading the file that contains the PASCAL-L machine code, locating variables, starting analysis, and writing the Jasmin code to a file. At the end of each stage, the user can scroll up and down the display area to inspect the intermediate output. We show below a simple example that illustrates the translation process. The original PASCAL-L program is

```
program proctest;
var i, , k, t : integer;
procedure Sort;
begin
   writeln('Inside procedure Sort: ', i, j, k);
if i > j then
```

```
        t := i; i := j; j := t;
elsif k < i then
        t := k;  k := i; i := t;
elsif k < j then
        t := k; k := j; j := t
fi;
writeln('The result after sorting is: ', i, j, k);
   end;
begin
writeln('Enter three integers : ');
readln(i, j, k);
writeln('Before calling procedure Sort : ', i , j, k);
Sort;
writeln('After calling procedure Sort : ', i, j, k);
end.
```

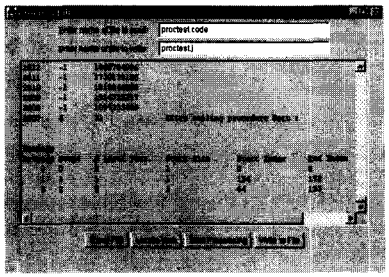
The translation process starts with the compiled code generated by the PASCAL-L compiler that is partially shown in [Fig. 7] when the "Read File" button is clicked.

The next step is to identify variables, constants, labels, and methods. [Fig. 8] shows the variable keys and labels found in the code. The summary is shown in [Fig. 9]

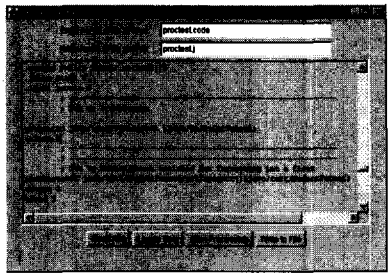


[Fig. 7] Partial code generated by the PASCAL-L compiler

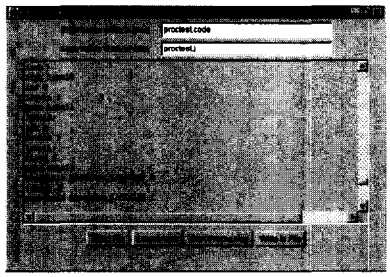
[Fig. 8] Variable keys and labels identified.



[Fig. 9] Extracted information about methods



[Fig. 10] Jasmin code for a method



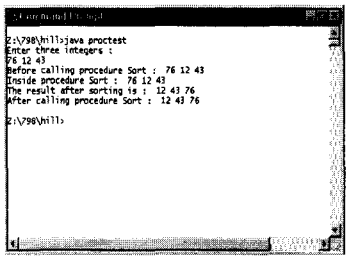
[Fig. 11] Partial bytecode of the if statement

Note that method 0 and some of the variables in [Fig. 9] are not in the original PASCAL-L program but they are default values in the Jasmin code.

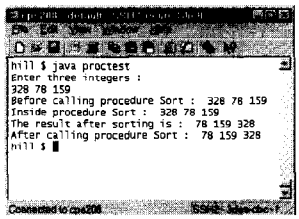
Then, PASCAL-L-Jasmin mapping is done by identifying the code patterns of the program structures and converting them to their Jasmin equivalent. The beginning of method2 is shown in [Fig. 10] and the portion of the bytecode for the if-statement is shown in [Fig. 11] After the Jasmin code is generated, the Jasmin assembler converts it into Java bytecode by the command

```
jasmin  proctest
```

The result is in a class file named proctest.class. Now, we can run the code just like running any Java program, illustrated in [Fig. 11] (on NT) and [Fig. 12] (on Solaris). It is clear that the translation did achieve its goal of making the PASCAL-L program "platform-independent."



[Fig. 11] Running the converted bytecode on NT



[Fig. 12] Running the converted bytecode on Solaris

We have tested over 30 PASCAL-L programs of all the features we discussed in the paper. All these programs worked out correctly with JVM. We are confident that the translation process is successful in meeting the following criteria:

(1) The analysis stage completes successfully with all variables and constants discovered and typed, except in cases where a failure to locate and type a variable or constant will not affect the outcome of the program.
(2) The pattern mapping stage finishes successfully with no sections of machine code left unmatched.
(3) The Jasmin assembler assembles the result of the mapping into a Java class file with no syntax errors discovered.
(4) The Java bytecode verifier finds no inconsistencies in the generated class file.
(5) The program runs in the JVM and produces results equivalent to the results of the original CMPU program.

Even though the example shown is quite simple, it still provides a glimpse into the challenges that will be faced by anyone translating architecture-dependent programs into Java programs.

7. Conclusion and Discussion

A binary-to-binary translation system is presented in the paper. Although it is for translating the binary code of the particular language PASCAL-L, the concept and methodology are quite general and may applied to other procedural languages. The assumption is that we do have the knowledge of the internal format of the machine code. This assumption is quite natural because the machine code is the input to the translation process.

The very basic idea of the translation process is the analysis of the code patterns to identify the language structures, particularly variables and subprograms. Since the machine code does not include the identifiers of the items (variables, subprogram names, named constants, type names, etc.), they can be recognized only through the opcodes. Fortunately the code patterns of different program structures are different, although some share sub-patterns. This may also be the case for the code patterns of many of the procedural languages.

While many differences exist between a source language and Java, it is possible to map the machine code of the source language into Java bytecodes that fully comply with the requirements of the Java virtual machine with little loss of accuracy. In doing so, a non-object-oriented program becomes a completely object-oriented program. A platform-dependent program becomes platform inde-

pendent with all register and memory references removed and replaced with an abstract stack representation. It is our belief that this type of software will play an important role in system integration by filling the void of integration at the binary level.

We are currently working on enhancements of the system so that is could be incorporated in the original compiler with optimization.

※ References

[1] Joshua Engel, "Programming for the Java Virtual Machine," Addison Wesley Longman, Inc., 1999.

[2] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, "The Java Language Specification," 2nd Edition, Addison Wesley Longman, Inc., 2000.

[3] Michael Gschwind, Erik Altman, Sumedh Sathaya, Paul Ledak, David Appenzeller, "Dynamic and Transparent Binary Translation Computer," IEEE Computer Society, Vol. 33, No. 3, 2000, pp. 54-59.

[4] Gongzhu Hu, "Theory and Practice of Compiler Construction," CMU Printing Services, 2000.

[5] Qiaoyun Li, "Java Virtual Machine - Present and Near Future." Proceedings of TOOLS-26'98, IEEE, August 3-7, 1998.

[6] Tim Lindholm, Frank Yellin, "The Java Virtual Machine Specification," 2nd Edition, Addison Wesley Longman, Inc., 1999.

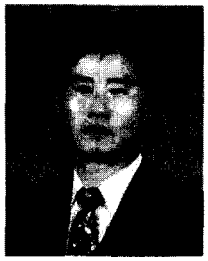
[7] Jon Meyer, Troy Downing, "Java Virtual Machine," O'Reilly and Associates, Inc., 1997.

[8] Bill Venners, "Inside the Java 2 Virtual Machine," The McGraw-Hill Companies, Inc., 1999.

[9] Cindy Zheng, Carol Thompson, "PA-RISC to IA-64: Transparent Execution, No Recompilation Binary Translation," Computer, IEEE Computer Society, Vol. 33, No. 3, 2000, pp. 47-52.

[10] JavaTM 2 SDK, Standard Edition Documentation, Sun Microsystems, Inc., 2000, http://java.sun.com/products/jdk/1.2/docs/index.html

강 전 근



1977년 동국대학교 전자공학과 졸업(공학사)
1985년 한양대학교 대학원 전자계산학과 졸업(공학석사)
1997년 대구가톨릭대학교 대학원 전자계산학과 졸업 (이학박사)
1979년 -1985년 : 한국전력공사 정보처리처
1985년 - 현재 영진전문대학 컴퓨터계열부 교수
관심분야 : 데이터베이스, 인공지능, 소프트웨어공학

김 행 곤

1985년 중앙대학교 전자계산학과 졸업(공학사)
1987년 중앙대학교 대학원 전자계산학과 졸업(공학석사)
1991년 중앙대학교 대학원 전자계산학과 졸업(공학박사)
1978～1979년 미 항공우주국 객원연구원
1987～1989년 AT&T 객원 연구원
2000.12～2001. 현재 미 Central Michigan University 교환교수
1990～현재 대구가톨릭대학교 컴퓨터공학부 부교수
관심분야 : 객체지향 시스템 설계, 사용자 인터페이스, 소프트웨어 재공학, 유지보수 자동화 툴, CASE, 소프트웨어 컴퍼넌트 공학