Programming Language Translation

Lili Qiu

lqiu@cs.cornell.edu

Department of Computer Science

Cornell University

Ithaca, NY 14853

Abstract

As programming languages become more and more diversified, there is an increasing demand to translate programs written in one high-level language into another. Such translation can help us more effectively reuse the existing code, especially when automating translation is possible. However due to many subtle distinctions between different languages, usually only a subset of translation can be automated. The first half of the paper describes the details of automating most of the translation from C to C++, as well as the difficulties encountered. The second half of the paper talks about the experience of manually porting Java programs to C++, and identifies some of the issues and challenges in automating this translation process. Through the discussions, it is evident that translation is heavily language specific. Comprehensive knowledge about the languages and their subtle distinctions is essential. On the other hand, designing tools to allow high level specification of translation rules and effectively incorporate human interaction is a generic approach to any language translation problem, which is an interesting research problem to explore.

Keywords: C, C++, Java, lex, parse, software reuse, translation, yacc.

1 Introduction

"Software reuse" is the key to improving both the quality of software and the productivity of software engineers. Reuse can take many forms - reuse of specification, designs, architecture, and code [8]. This paper focuses on a form of code-oriented software reuse, more specifically, translation of programs written in one high-level language into another.

The demand of such translation is growing as programming languages get more and more diversified. However translating these codes manually is costly and error-prone. Automatic translating tools are highly desirable. There have been research in high-level language translation since early 80's, such as translating from Cobol into Hibol [3], from RPL to PL.8 [7], and more recently from Java to C [6] [10]. This paper talks about the translation from C to C++, and from Java to C++.

The first half of the paper describes our experience of building the translator converting C program into C++, and the difficulties encountered. The second half talks about the translation from Java to C++, and identifies the issues it involves. Throughout the discussions, it is evident that translation is heavily language specific. Comprehensive knowledge about the languages and their subtle distinctions is essential. On the other hand, designing tools to allow high level specification of translation rules and effectively incorporate human interaction is a generic approach to any language translation problem, which is an interesting research problem to explore.

2 Translation from C to C++

Our need of translating from C to C++ arises from building a network simulator. There are two major requirements for building the simulator: First, the simulator must be able to simulate network in a realistic way. To achieve this, we decide to use FreeBSD kernel source code (which is written in C) as our code base, because it is the basis for most UNIX kernels and NT kernel. Second, the simulator should be able to simulate multiple end hosts, each of which has a FreeBSD kernel running on top of it. To achieve this, we need to have a mechanism to encapsulate the kernel information within each host. There are two typical approaches for this: i) multiple address space implementation, where the kernel of each host runs as a user level process; ii) single address space implementation, where the kernel of each host is simulated as a thread. The multiple address space approach is inefficient due to significant overhead for context switch and process creation/deletion. So we favor the single address approach. However, we have to resolve the name space conflict due to multiple instances of the kernel in a single address space. A natural solution is to use object-oriented approach. We can treat each host as an object, and encapsulate the information of each host within the host. We choose C++ as the language to implement the entire simulator, because it is object oriented, and closely related to C. Due to the large amount of C source code, it would be infeasible to port it to C++ manually. It is highly desirable to have some tools that can automatically translate C to C++. Unfortunately, to our knowledge, there is no such system or tools available. The only related work we know so far is [8], but it mainly focuses on how to generalize C functions into C++ function templates.

2.1 Comparison between C and C++

Although C and C++ share lots of commonality, their distinction is significant enough to make entirely automating translation impossible.

2.1.1 Different Programming Paradigms

C is a procedural language, whereas C++ is an object-oriented language. This difference is so radical in that objects differ fundamentally from procedures in their semantics, composition mechanism, and structuring mechanism. As any object-oriented programming, the first question to answer in converting from C to C++ is what design decision to make, that is what kind of object needs to be created, how to organize the relationship between various objects. Coming up with a good object model to sincerely reflect the structure being modeled is by no means a mechanical process but an art.

However since our need is to avoid name space conflicts in C, the elegant and efficient class structure is not critical. Instead we favor the simplest possible solution that requires minimum modification. The two most natural strategies are either to map each of the C files into a C++ class, or to combine all the C files into a single C++ class. The former solution does not have global variables naming conflicts as exist in the latter case. However it has its own complications: the interactions between objects of different classes can be quite complicated. Every invocation of a function requires checking which object the function belongs to. To remove such inefficiency, we chose to have a single C++ class for the entire program. In this approach all the global variables and functions in C become member data and member functions of a C++ class respectively.

2.1.2 Strongly vs Weakly Typed

C++ is strongly typed language as opposed to C. The implicit type conversion in C++ is more restrictive. It is thus necessary to detect all the implicit type conversions in the original C program that are not permitted in C++, and make them explicit. In particular, type checking is required for every assignment to see if the two sides share the same type or obey the rule of implicit type conversion. Since the right side can be any kind of expression, which can involve a function, a symbol table of all functions and variables is necessary to facilitate the type checking.

Function pointer further complicates the issue. In non ANSI C, $g()$ means g can take arbitrary number of arguments of any type, whereas in C++ it means g takes no arguments. Therefore to properly converting a function pointer from C to C++ requires dynamically determining its type and explicitly converting the function pointer to the one with the right type. For example, if in C the declaration and invocation look as follows:

```
/* declaration */
int *f();

/* invocation */
(*f)(a,b); /* where a and b are both integers */
```

then its corresponding C++ version should be:

```
/* declaration */
typedef int *F_P(int, int);

/* invocation */
F_P g=(F_P)f;
(*g)(a,b);
```

In the case when $f()$ is a member function of a class C, the correct C++ conversion should be:

```
/* declaration */
typedef int (C::*MF_P)(int, int);

/* invocation */
MF_P g=(MF_P)f;
(this->*g)(a,b);
```

Thus in order to determine the function prototype dynamically, the type of each parameter should be extracted properly using the symbol table.

Another solution to this problem is to use "...", which avoids determining the types of arguments, though not a good practice. For the above example, the possible C++ version can be the following:

```
typedef int (C::*MF_P)(...);
MF_P g=(MF_P)f;
(this->*g)(a,b);
```

The weakly typed feature of C is also reflected through functions' return values. More specifically, some functions in C do not return anything, even though they are defined as *int*. C++ compiler will complains about this. But what value to return can be context dependent. For example, by convention, 0 is returned upon normal exit, and the corresponding error number should be returned in case of error. To determine a correct return value requires understanding of the code, which seems impossible to automate. On the other hand, it is unlikely that the returned value will be used, since otherwise C program would not have left it undefined. Therefore the translator can actually return anything it thinks reasonable. It can even always return the same value whenever there is a missing return value.

2.1.3 Static Variable

The keyword *static* has different meaning in C and C++. In both C and C++, declaring a local variable as static allows the variable to retain its previous value when the block is re-entered. However another meaning of static in C is to provide privacy so that the scope of a static variable is only the remainder of the source file in which it is declared [9]. It is thus unavailable to functions defined earlier in the file or to functions defined in other files, whereas in C++, declaring the member as static ensures that there will be only a single copy of it. This distinction suggests that in C++ we can no longer rely on using the keyword *static* to prevent naming conflicts, since all the non-local variables in the original C files become data members visible in scope of the entire C++ class. Renaming the conflicting static variables is thus necessary.

2.2 Implementation of Semi-automatic Translator

The role of the translator is to parse C programs and generate the corresponding C++ codes. Since not all the translations can be automated, my goal is to automate as much translation as possible. I used *cproto* [5] as the starting point for building the translator. It is a program that generates function prototypes and variable declarations from C source code. It can also convert the function definitions between the old style and ANSI C style. But to avoid implementing the entire C language grammar, no processing is done to the function bodies.

I modified *cproto* to suit our needs. As in *cproto*, *lex* and *yacc* are used to generate lexical analyzer and syntactical parser for C language. To automate as much translation as possible, more thorough language parser has been built. The appropriate semantic actions are carried out to produce desirable C++ code. In particular, suppose the name of the class to be generated is foo, then the output of the translator are three files: *include.h*, $foo.h$, and $foo.c$:

1. include.h has $\#include$ and $\#define$

2. foo.h has the class definition. More specifically,

- All non-local variables in C are considered member data of that class in C++;
- All functions in C become member functions of that class in C++. Thus the prototypes of all functions are included in the class definition. Converting the prototypes to ANSI C is necessary in the case when the original prototypes are non ANSI C style.
- Declaring all the member functions and data as public offers more flexible access, and safety is not compromised since original C programs do not impose access restriction.
- All the type definitions are also included here.

3. foo.cpp contains the bodies of all the member functions. In particular, the following modifications are made in the conversion:

   - prepend class name to all the member functions
   - generate the constructor, which contains the initialization for all the global variables that are intialized in the original C program during declaration
   - generate the main function shown below:

```
//global function
int main (int argc, char *argv[])
{
 foo *obj=new foo();
 // call the main() in the original C program, which has been converted to
 // a member function
 obj->main(argc, argv);
 return 0;
}
```

2.2.1 Details

1. Conflicting global variables

   The conflict naming of global variables can happen in the following three cases:

   (a) A static variable defined in one file has the same name as a static variable declared in another file

(b) A static variable defined in one file has the same name as a non-static variable declared in another file, and the non-static variable is not used in the file in which the static variable is defined

(c) A static variable defined in one file has the same name as a non-static variable in another file, and the non-static variable is used in the file in which the static variable is defined (The non-static variable can be used only before the declaration of the static variable.)

Having identified the possible conditions in which conflicts can take place, I used C program along with some perl scripts to facilitate the renaming.

2. Initialization

Since all the global variables in the original C program are changed to be member data of C++ class, and member data are forbidden to be initialized inside the class definition according to ANSI C++ specification, the initialization of these variables cannot be done at the same time of declaration. Instead all the initializations should be placed in the constructor. The initialization constant can be primitive type, such as *int*, *char*, *double*, or aggregate, like *struct* and array. Initializing a variable with an array requires special treatment, because using such form of initialization is only permitted when the variable is being declared. For example, the following code

```
int a[]={1,2,3,4,5};
```

cannot be converted into:

```
//foo.h
class foo {
  ...
  int a[];
  ...
};

//foo.cpp
foo::foo() {
  a[]={1,2,3,4,5};
}
```

Two things need to take care of:

- The size of the array should be determined during the declaration;
- The initializing assignment is no longer legal. One solution, which is implemented in my translator, is to declare a new local variable of the same array type with the right initialization, and then use a *for* loop to copy each of its elements to the member data. This results in the following translation for the above code:

```
//foo.h
class foo {
  ...
  int a[5];
  ...
};

//foo.cpp
foo::foo() {
  int local_a[]={1,2,3,4,5};
  int i;
  for (i=0; i<5; i++)
      a[i]=local_a[i];
}
```

Having taken care of the above differences as well as many other details, I finally have a semi-automatic translator. I have used it to translate some FreeBSD codes, such as *ifconfig* and ethernet interface. I am planning to use it to translate even larger applications including *gated*.

2.3 Usage of Translator

The translator is semi-automatic, and requires human involvement in using it. Below are the procedures to do the translation:

1. Rename all the conflicting global variables via perl scripts (Possible conflicts are identified in the subsection 3 above.);

2. Use my translator to generate a single C++ class from a C program which can consist of any number files;

3. Manually modify the C++ class produced by the translator. Since the translator does not automate every necessary translation, there are errors in compiling the automatically generated C++ codes. So the goal of this step is to remove all the compilation errors. Since translation is syntactical, as soon as compilation errors are eliminated, the program is ready to run.

3 Translation from Java To C++

Translation from Java to C++ is another interesting example to look at. One of the major reasons for this translation is for performance improvement. The following discusses some problems during the translation. Though the list is by no means exhaustive, it gives some ideas of what need to be taken care of during the translation.

3.1 Comparison between Java and C++

Both Java and C++ are object-oriented languages, so there exists a direct map from Java class to C++ class. The class structure problem, which is hard in the conversion from C to C++, becomes trivial here. However the significant distinctions between Java and C++ still makes the translation very difficult.

3.1.1 Garbage Collection vs. Explicit Deallocation

Java does garbage collection, so that Java programmers are free to allocate as many objects as they like without worrying about deallocation. In contrast, C++ requires explicit deallocation. To properly convert Java programs into C++, we have to either do explicit deallocation, or use garbage collection. Explicit deallocation is good from the performance point of view. However to automatically detect where it is safe to do deallocation is very tricky. Garbage collection algorithms are more suitable for automating the translation. There are a number of garbage collection algorithms. The most common algorithm is reference counting, but it has problem in dealing with cyclic structures. The mark-sweep is another commonly used algorithm: A mark-sweep garbage collector traverses all reachable objects in the heap by following pointers beginning with the "roots", i.e. pointers stored in statically allocated or stack allocated program variables. All such reachable objects are marked. A sweep over the entire heap is the performed to restore unmarked objects to a free list, so they can be reallocated. The Boehm-Demers-Weiser garbage collector [1] is designed for C and C++ based on mark-sweep algorithm. It is well-suited in facilitating the conversion from Java to C++ due to its conservative collection.

3.1.2 Interface and Abstract Class

Java uses interface to avoid supporting "multiple inheritance" of method implementations from more than one superclass. Though named differently, C++'s abstract class is somewhat equivalent to Java's interface. So special care needs to be taken when converting Java's interface declaration and implementation. For example, the following Java code:

```
// declare interface
public interface a {
  public void set(int state);
  public int  get();
}

// implement interface
public class b implements a {
  private int state;
  public void set(int state) { this.state=state;}
  public int  get() {return this.state;}
}
```

can be converted into the following C++ codes:

```
// declare abstract class
// containing only pure virtual functions
class a {
  public:
    virtual void set(int state)=0;
    virtual int  get()=0;
}

// extend the abstract class
```

```
class b : public a {
  private:
    int state;
  public:
    void set(int state) { this.state=state;}
    int  get() {return this.state;}
}
```

3.1.3 Different Parameter Passing Mechanism

In Java, all the primitive data are passed by value, and all objects are passed by reference, whereas in C++ by default all the parameters are passed by value. Therefore the translator needs to take care of this as follows: pass to the function the pointers of objects instead of objects themselves, and the function body needs to change accordingly to take into account the actual parameters are pointers.

3.1.4 Default Value vs. Garbage Value Before Initialization

Since all variables in Java have default values before initialization, whereas uninitialized variables in C or C++ have garbage values, the conversion needs to initialize all variables properly before using them.

3.1.5 Data Type

Java's data types are very similar to C++'s, but there exist some subtle differences. The most intricate one is that Java explicitly defines size and signedness of every data type, whereas C++ leave them as platform dependent. For example, *long* in Java is 64 bits. The corresponding data type in C++ on Solaris is *long long*, while the equivalence on NT is *__int64*. Thus converting from Java to C++ requires picking up the right data type based on the knowledge of the platform on which the application will run. If data type is chosen inappropriately, the program may fail to run. For instance, the length of UDP packet can be computed incorrectly if the data conversion is not properly taken care of. However choosing the correct data type can be a tedious task given the large number of different platforms. Creating mapping from Java data types to C++'s for all the platforms the application is expected to run is indispensable step. Once we have manually come up with a table of the mapping, the translation can then automatically take care of data type conversions through table lookup.

3.1.6 Java Built-in Packages

Java has very rich built-in packages, ranging from GUI and utility to network and multithreading. The equivalence in C++ is not immediately available. Thus it is necessary to produce these equivalent classes either through translation or through direct implementation. Translation seems appealing, for it allows reusing the codes to save some work, especially when automatic translation is possible. However automatically converting these built-in packages into C++ is not always possible, because Java source codes do not include all the implementations. Instead some of the implementations are embedded inside the Java Virtual Machine. Therefore the more viable approach is to divide the built-in classes into two groups: those that include all the implementation at the source code level and those that do not. For the former group we can apply automatic translation as much as possible, while for the latter group we have to manually implement these Java built-in classes in C++, which can be operating system dependent. C++ libraries can be used to facilitate the manual conversion. After this is done, we then have a new C++ library that contains equivalent C++ version of all the Java built-in classes. At this point, translating a Java application program only requires translating the application program itself, since the interfaces of Java built-in classes and equivalent C++ version are the same. Nevertheless building C++ version of Java built-in class is a very time-consuming and complicated task.

3.1.7 Other Distinctions

- *Exception*

  Both Java and C++ support *Exception*. The only difference is that C++ does not have *throw* clause in the function prototype.

- Virtual Function

  All the functions in Java are virtual as opposed to C++, where the virtual functions need to be explicitly defined so. One obvious solution is to explicitly define all the functions as virtual in C++ during the translation.

- The *package* and *import* statements

  The *package* statement specifies the package that the classes in a file of Java source code are part of, and the *import* statement allows us to refer to classes by abbreviated names [4]. Since C++ has no notion of package, the *#include* can be used as a substitute though not exactly the same meaning.

4 Conclusion

Programming language translation is becoming increasingly important due to either technical reasons or the market demands. Automating these translations is an appealing idea. However it is very challenging to build an (semi)automatic translating tools. Even though C, C++, and Java are so similar to one another that most codes do retain in the translated version, the translation process as shown above are not straight forward at all. Lots of issues and language details need to be addressed, and manual operations are inevitable.

Using *yacc* generated parser helps a lot in building a translator. However the semantics action is still specified at very low level. It will be much more useful to have tools, which allow high level specification of translation rules and effectively incorporate human interaction. Designing such tools is a generic approach to any language translation problem, and is also very helpful to other kind of program reuse. This is an interesting research problem to explore.

ACKNOWLEDGEMENTS

The author likes to thank Professor S. Keshav and Yin Zhang for their valuable comments on the paper.

References

[1] The Boehm-Demers-Weiser garbage collector. http://reality.sgi.com/boehm_mti/gc.html.

[2] J. M. Boyle and M. N. Muralidharan. "Program Reusability through Program Transformation". IEEE Transactions on Software Engineering 10, 5 (Sept. 1984).

[3] G. G. Faust. "SemiAutomatic Translation of Cobol into Hibol". Master Thesis. Massachusetts Institute of Technology, 1981.

[4] D. Flanagan. "Java In a Nutshell". O'Reilly.

[5] C. Huang and T. Dickey. *cproto* man page, http://www.sdsc.edu/~lindsey/CMDA/docs/cproto.html.

[6] JCC. "Java to C Converter". http://www.ph-erfurt.de/information/java/dev/misc/jcc/.

[7] T. R. Kennedy, III. "Using Program Transformations to Improve Program Translation".

[8] M. Siff and T. Reps. "Program Generalization for Software Reuse: From C to C++". In *Proc. SIGSOFT '96*, 1996.

[9] B. Stroustrup, "The C++ Programming Language", 2nd Edition, Addison Wesley.

[10] Toba. "Toba: A Java-to-C Translator". http://www.cs.arizona.edu/sumatra/toba/.

[11] R. C. Waters. "Program Translation via Abstraction and Reimplementation". A.I. Memo 949, Massachusetts Institute of Technology, Dec. 1986.

[12] P. R. Wilson, "Uniprocessor Garbage Collection Techniques", 1992 International Workshop on Memory Management.